Сергей Коржинский

Изучаем Delphi

Практическое пособие для изучения
программирования в среде Borland Delphi.

Электронная редакция

Версия 1.0

Москва, 2007.

© Коржинский С.Н. Все права защищены.

http://www.snkey.net

Предисловие к электронной редакции

Книга, на которую вы, скорее всего, смотрите с экрана своего компьютера, изначально готовилась к публикации в обычном, «бумажном» варианте. Однако процесс написания и подготовки довольно сильно затянулся, в и тоге было решено выпустить материал без окончательной корректуры, «как есть» в виде электронной версии. Объем книги составляет приблизительно 340 страниц стандартного для компьютерных книг формата 70x100/16, в электронной же версии книга состоит из PDF-файла и архива с примерами.

Поскольку написание книги по Delphi для автора этих строк – хобби а не желание что-то заработать, то, выпуская эту «рукопись» в свободное плавание по просторам всемирной сети, остается только порадоваться за то, что любой заинтересованный в изучении такой замечательно среды разработки, как Delphi, сможет совершенно свободно, бесплатно, и легально(!) воспользоваться моими трудами.

ВНИМАНИЕ

Запрещается коммерческое использование данного материала, а именно публикация книги на каком-либо физическом носителе с целью продажи, равно как и платное распространение какими-либо иными способами без письменного согласования с автором

Введение

Настоящая книга, посвященная популярной среде программирования Delphi, ориентирована на начинающих разработчиков, желающих самостоятельно освоить этот замечательный инструмент разработки приложений. Поскольку лучший способ изучения данного предмета – это его непосредственное использование, то изложение материала во многом будет опираться на примеры. При этом автор стремился сделать книгу не просто информативной, но и достаточно интересной.

Изложение материала, несмотря на достаточную простоту и ориентированность на начинающих разработчиков, базируется на фундаментальной концепции изучения среды программирования. Иначе говоря, сначала изучается собственно основы программирования в контексте языка Object Pascal, используемого в Delphi, и лишь затем – визуальные средства разработки и компоненты интерфейса Windows. Возможно, такой подход и не позволит вам создать «настоящую» Windows-программу через пару минут после того, как вы откроете эту книгу, но по ее прочтении вы сможете самостоятельно создавать гораздо более интересные и полноценные проекты, не прибегая к дополнительным источникам. Автор уверен, что такой подход в итоге окажется гораздо более продуктивным, чем в том случае, если бы книга ограничивалась указаниями вроде «поставьте эту кнопку сюда, и введите такой код туда».

В итоге мы видим перед собой книгу, состоящую из нескольких частей. В первой части рассказывается о Delphi и об основах программирования в этой среде. Впрочем, если вы когда-нибудь решитесь сменить среду разработки, то накопленные вами знания все равно пригодятся вам, будь то другой вариант Pascal (хотя бы даже Kylix), или вообще другой язык – C++, C#, или Java.

ПРИМЕЧАНИЕ

> Следует отметить, что использованный в книге подход не ограничивает вас какой-либо конкретной версией Delphi. За основу для изображений среды была взята «промежуточная» 7-я версия, однако все это будет работать как в более ранних, так и более поздних версиях (без лишних оговорок можно смело взять диапазон от Delphi 6 до Delphi 2006).

Вторая часть вводит вас в мир ООП – объектно-ориентированного программирования. Что это такое, вы узнаете еще в первой части, а во второй – научитесь использовать на практике. Кроме того, вторая часть познакомит вас с программированием непосредственно для Windows, в частности, будут рассмотрено взаимодействие с функциями это операционной системы – Windows API.

В третьей части книги мы вплотную займемся изучением пользовательского интерфейса, вернее – вопросом его создания при помощи визуальных средств Delphi посредством ее замечательной библиотеки компонентов – VCL. Разумеется, это не значит, что в предыдущих двух частях (т.е., фактически, прочитав половину книги), мы не создадим ни одной Windows-программы – создадим, и не одну. Но детальное изучение визуальных компонентов было отложено именно потому, что без знания языка программирования Delphi и ООП попросту невозможно понять, как это работает, а не попытавшись сделать что-то средствами Windows API, трудно понять, почему это так необходимо.

Наконец, четвертая часть настоящего издания посвящена извечной теме разработки баз данных. Ведь, на самом деле, подавляющее число разработчиков, и, прежде всего – использующих Delphi программистов, как раз и занимаются разработкой баз данных, а вовсе не написанием компьютерных игр или различных «офисов», «мозилл» и «линуксов».

ПРИМЕЧАНИЕ

> Впрочем, всякий настоящий программист стремится создать что-нибудь «большое и светлое», хотя бы даже в свободное от основной работы время. Именно отсюда и берутся те самые «линуксы» и «файрфоксы».

В заключение будет рассмотрен еще один животрепещущий аспект современного программирования – создание программ для Интернета. При этом мы возьмем за основу не классический вариант ActiveX-элементов, что было актуально во времена Delphi 3, а истинные VCL-компоненты Internet Direct (Indy), впервые появившиеся в Delphi 6, и доступные отдельно для более ранних версий Delphi (4-й и 5-й) в виде бесплатных наборов компонент с открытым исходным кодом.

Таким образом, книга охватывает практически все аспекты применения Delphi, по крайней мере, в тех рамках, что может себе позволить ориентированный на начинающего разработчика самоучитель.

Удачи вам в изучении и приятного программирования!

Часть I.
Основы программирования в Delphi

О программировании и о Delphi

Начнем наше ознакомление с программированием в среде Delphi с таких вопросов, как суть собственно программирования, его основы и подвиды. Также мы рассмотрим, что представляет собой среда Delphi, чем она отличается от других, и какой путь был ей пройден с момента появления 10 лет назад.

Что такое программирование

Итак, прежде всего, уясним для себя, что же понимают под термином программирования? Сейчас многие называют себя программистами, не зная даже о том, что это такое: на проверку оказывается, что человек, изменивший пару строк кода в HTML-документе уже готов считать себя программистом. Но на самом деле, работая с HTML (именно с самим HTML), стать программистом нельзя, поскольку HTML, что явствует из названия (Hypertext Markup Language – язык *разметки* гипертекста), языком *программирования* не является.

В результате мы имеем первое условие: для изучения программирования нужен какой-либо язык программирования. В нашем случае это будет язык Object Pascal последних версий (14.0 или 15.0).

Далее, допустим кто-то, называющий себя программистом, не только правил пару строк кода HTML, но и видел Delphi, и даже нарисовал в нем форму с 2-3 кнопками и текстовым полем. Это ли программист? Скорее всего, тоже нет. Дело в том, что под визуальной оболочкой, на самом деле, скрывается сложнейший механизм, обеспечивающий взаимодействия элементов управления (тех же кнопок) с пользователем и операционной системой – с одной стороны, и позволяющий выполнять практически любые вычислительные действия – с другой. При всем этом следует уметь делать главное – составлять алгоритмы поведения программы – будь то ее реакция на нажатие той или иной кнопки, или создание функции для решения сложного математического уравнения.

Так вот, программирование в, своем современном понятии, включает в себя знание принципов работы операционной системы, взаимодействия ее компонентов и внутреннего устройства. Ну и при этом сохраняется классическое определение программирования, а именно – умение составлять алгоритмы, математические и поведения программы. Все это реализуется в контексте какого-либо языка программирования, например, того же Object Pascal из Delphi.

Что такое Delphi

Теперь мы знаем, что программирование – есть составление алгоритмов плюс использование языка программирования. Но если изучаемым нами языком является Object Pascal, то что такое Delphi? Ответ таков: Delphi – это RAD, или Rapid Application Development – среда быстрой разработки приложений. Иначе говоря, Delphi включает в себя не только все средства для работы с языком Object Pascal (тем более,

что начиная с версии, 2005 в Delphi имеется поддержка других языков программирования, в частности, C#), но и ряд дополнительных средств, призванных максимально ускорить и упростить создание программ. К таковым средствам относятся, прежде всего, визуальный редактор форм, при помощи которого за считанные минуты и без лишних усилий можно создать полноценно выглядящую программу, а так же прочие составные части визуальной составляющей разработки программ.

В результате время разработки программ существенно сокращается, поскольку отпадает необходимость просчитывать вручную расположение каждого элемента пользовательского интерфейса.

Вкупе с развитыми средствами для написания и отладки кода – специализированным текстовым редактором, оптимизирующим компилятором и отладчиком, Delphi являет собой средство быстрой разработки приложений. При этом основой Delphi для конечного пользователя является IDE – Integrated Development Environment (интегрированная среда разработки), которая объединяет в себе редактор кода и средства визуальной разработки, а также связывает это с компилятором, средствами разработки баз данных и прочими составными частями Delphi.

Delphi и другие

Разумеется, Delphi – не единственная среда быстрой разработки приложений. Существуют и другие RAD, столь же удобные для визуальной разработки программ, например, Visual Basic. Но BASIC известен как не самый мощный и удобный язык программирования, кроме того, программы на нем отличаются сравнительно невысоким быстродействием.

Вместе с тем, имеются и не менее мощные, по сравнению с Object Pascal, языки программирования, имеющие свои RAD – прежде всего, это язык C++ вместе с таким наиболее известными средствами разработки, как Visual C++ и C++ Builder. Однако C++ не в лучшую сторону отличается от Object Pascal с точки зрения простоты изучения. Кроме того, сборка и отладка программ на C++ происходит сравнительно медленно, хотя готовые программы выполняются так же быстро, как и созданные при помощи Delphi.

ПРИМЕЧАНИЕ

> Отметим, что в момент своего появления в 1995 году, Delphi была действительно уникальной средой, объединяющей в себе визуальное программирование с оптимизирующим компилятором и средствами для работы с базами данных.

В результате мы получаем практически идеальный продукт для быстрой разработки программ: с одной стороны простота и удобство, сравнимые с Visual Basic, а с другой – мощь, скорость и гибкость, характерные для C++.

Здесь следует отметить, что используемый в Delphi язык Object Pascal имеет мало общего со своим прародителем – языком программирования Pascal, который, возможно, вы изучали в школе или в ВУЗе. Прежде всего, в Delphi применяется чистая объектно-ориентированная модель программирования, в то время как Pascal был обычным процедурным языком. В чем же суть отличий процедурного программирования от объектно-ориентированного? Вот этот вопрос мы и обсудим в следующих двух параграфах.

Процедурное программирование и алгоритмы

Практически все современные языки программирования являются процедурными по своей сути. Это означает, что каждая программа состоит из набора процедур, каждая из которых решает ту или иную задачу, при этом одна процедура может вызывать любую другую, в том числе и саму себя (последнее, т.е. вызов процедурой самой себя, называется рекурсией). Собственно программа составляется на основе алгоритмов, причем достаточно всего трех их разновидностей – выбора (развилки), повтора (цикла) и перехода. Впрочем, последний тип на практике практически не используется ввиду своей дурной репутации (речь о GOTO – некогда любимом, от безысходности, операторе BASIC -программистов).

Давайте рассмотрим простейший алгоритм – для игры «угадай число». Логика его работы состоит в том, что требуется проверить, является ли число большим или меньшим, чем загаданное, и если это так, ты вывести соответствующую подсказку, а если оно не больше и не меньше (т.е. совпадает) – то игра заканчивается. Блок-схема такого алгоритма будет состоять из 2 ветвлений (рис. 1.1).

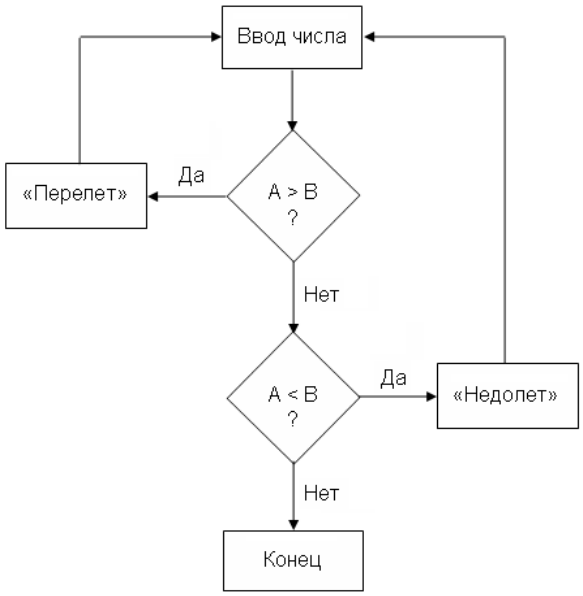

Рис. 1.1. Блок-схема простого алгоритма

Первым делом здесь у пользователя запрашивается число A, которое дважды сравнивается с предварительно заданным числом B. Сначала проверяется, не является ли число A больше, чем B, и если да, то программа сообщает, что произошел «перелет», и возвращается на точку ввода числа. Если число не больше, то происходит второе сравнение – на то, не является ли A меньше B. Если это так, то выводится сообщение о том, что число меньше («недолет») и, опять-таки, происходит возвращение к началу. Но если число A не меньше числа B, то, учитывая тот факт, что на данный момент уже известно, что оно и не больше, следовательно, оно равно числу B, и программа завершает свою работу. При этом она может вывести поздравление пользователя с тем, что он угадал число.

Данный алгоритм может быть частью другого алгоритма. Например, здесь не предусмотрена генерация самого числа, которое следует отгадать. Так же отсутствует возможность повторной игры. Чтобы исправить эту ситуацию, рассмотрим блок-схему алгоритма, управляющего выполнением программы в целом (рис. 1.2).

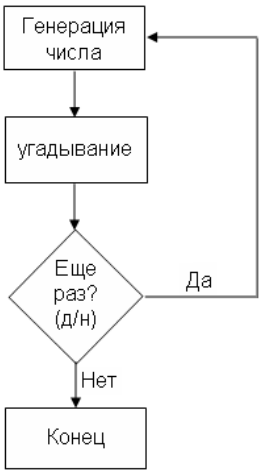

Рис. 1.2. Блок-схема управления

В начале выполнения генерируется новое число, которое будет отгадывать пользователь. Затем идет собственно блок угадывания, уже рассмотренный нами (см. рис. 1.1), – здесь он представлен в виде одного простого модуля, поскольку на ход выполнения программы в целом не влияет. По завершению этого блока мы имеем единственное ветвление – в зависимости от того, что ответит пользователь на предложение повторить игру. Так, если ответ будет положительным, то программа должна будет перейти к началу, т.е. к генерации нового числа. Если же пользователь отказывается, то работа программы должна быть завершена.

Конкретная реализация того или иного алгоритма зависит от применяемого языка программирования, а так же от предпочтений создателя программы, поскольку, как правило, одну и ту же конструкцию можно решить, используя несколько различных методов. Например, для возврата к какой-либо точке, обозначенной на блок-схеме, можно использовать как циклы, так и операторы безусловного перехода. Ну а сами блоки могут выноситься в отдельные процедуры и вызываться в качестве подпрограмм. Единственной общей чертой является условный оператор, отвечающий за развилку - практически во всех языках, и Pascal тут не исключение, он называется IF. Впрочем, с операторами Object Pascal мы еще успеем ознакомиться.

Об объектно-ориентированном программировании

Если выбор, повтор и переход – краеугольные основы любого процедурного языка программирования, то для объектно-ориентированного программирования важны такие понятия, как инкапсуляция, наследование и полиморфизм. Рассмотрим эти понятия, без которых ни один поддерживающий объекты язык не может называться объектно-ориентированным, подробнее.

ПРИМЕЧАНИЕ

> Под термином «объект» в программировании понимают некий сложный тип данных, к которому могут быть привязаны уникальные для данного типа свойства, а в случае ООП – также и методы.

Инкапсуляция представляет собой объединение данных и обрабатывающих их подпрограмм – методов – внутри одного объекта, называемого в ООП «классом». Это означает, что в классе инкапсулируется все необходимое для работы с тем или иным объектом.

Наследование – это еще одна важная составная часть ООП. Под наследованием понимают возможность создания типа (объекта), базирующегося на определении другого объекта. При этом все свойства и методы родительского объекта, могут быть унаследованы потомком. К примеру, если у нас есть объект «машина», то на его основе можно создать другой объект, скажем, «трамвай», который унаследует все его свойства, и получит вдобавок к ним какие-либо собственные. В свою очередь, этот объект так же сможет иметь своих потомков, и т.д. В результате образуется дерево объектов, называемое иерархией классов.

В Object Pascal все классы происходят от единого предка – класса TObject, который реализует такие общие для всех классов действия над объектом, как создание и удаление.

СОВЕТ

> В поставку коробочных версий Delphi включен плакат, на котором изображено дерево классов для всех основных компонент Delphi. Поэтому если вы – счастливый обладатель лицензионного продукта, то можете наглядно изучать устройство этой системы, разместив этот плакат на своем рабочем месте.

Наконец, третий кит, на котором стоит ООП – это полиморфизм, или возможность создавать в рамках родственных объектов одноименные свойства и методы, которые будут отличаться по своей сути. Скажем, у нас уже есть классы «машина» и «трамвай», и для обоих определено свойство «мотор», но если для машины тип мотора будет объектом одного типа, то для трамвая – совсем другого.

Еще одна важная составная деталь, характерная для ООП – это скрытие, которое позволяет делать невидимым ряд свойств объектов. Иными словами, если у нас может быть определен объект типа «машина», для которого, в свою очередь, определены такие свойства, как шасси, мотор и колеса. Так вот, если мотор и колеса, можно сделать видимыми (и доступными) повсеместно, где только доступен сам класс «машина», то шасси – нет. Соответственно доступ к такому свойству можно будет получить только в тех частях программы, в которых производится описание самого объекта «машина». Такие свойства называют защищенными.

Подробнее об объектно-ориентированном программировании будет рассказано во второй части настоящего издания.

Визуальное программирование и Delphi

Итак, Delphi являет собой среду, работающую с объектно-ориентированным структурным языком программирования Object Pascal. В дополнение к этому, Delphi явля-

ется еще и средой визуальной разработки. Иначе говоря, уже в процессе создания программы видно, как она будет выглядеть во время выполнения. Визуальное программирование в разы сокращает время, необходимое на создание пользовательского интерфейса для приложений Windows. До появления визуальных средств разработки, программирование под Windows было сопряжено с большими трудностями, поскольку требовалось практически вслепую вычислять расположение каждого элемента управления в окне приложения, что приводило к множеству ошибок (рис. 1.3).

Рис. 1.3. Классический и визуальный подходы к созданию интерфейса

То, что Delphi является визуальной средой разработки, наносит свой отпечаток и на язык программирования. В частности, в нем предусмотрены специальные средства для того, чтобы определенные свойства классов могли быть доступны уже на этапе разработки программы, а не только во время выполнения. Но самой важной частью всего этого великолепия является библиотека классов – VCL (Visual Component Library). Именно VCL скрывает все острые углы внутренностей Windows, и позволяет легко создавать приложения, не вступая в затяжную войну с обработкой системных сообщений, обратных вызовов, дескрипторов и указателей.

При всем этом, разумеется, никто не запрещает вам при надобности использовать функции Windows напрямую, в обход VCL. Впрочем, такое может понадобиться весьма нечасто, поэтому для начинающего программиста на Delphi, для написания Windows-приложений ничего, кроме VCL, не требуется.

Визуальные компоненты мы детально рассмотрим в 3-й части этой книги.

Версии Delphi и их отличия

В завершение вводной части рассмотрим различия между версиями Delphi, попутно изучив историю развития этой замечательной среды разработки.

Delphi 1. Вышедший незадолго до появления Windows 95, Delphi 1.0 был первым инструментом разработки приложений Windows (3.1), объединившим в себе оптимизирующий компилятор, визуальную среду разработки и мощные средства для работы с базами данных. Для языка Pascal это стало настоящим перевоплощением – он стал Object Pascal.

Delphi 2. Вышедшая через год 2-я версия Delphi предлагала все то же, но уже для 32-разряных версий Windows (95/NT), опять-таки став первым средством, сочетающим 32-битный компилятор, средства для работы с БД и визуальную среду разработки, поддерживающую OLE (а после выхода обновленной Delphi 2.01 – и ActiveX). Язык так же изменился, став ориентированным на 32-разрядное применение.

Delphi 3. Дальнейшее совершенствование Delphi по всем направлениям привело к выходу 3-й версии, имевшей расширенный набор инструментов для создания приложений благодаря поддержке COM и ActiveX. В этой версии также впервые был сделан акцент на возможности создания приложений для Интернета, а так же появился нетипизированный тип данных – variant.

Delphi 4. Еще одна ключевая веха развития. В этой версии было введено много нового и полезного для упрощения разработки приложений, включая новую среду с динамическими подсказками и т.д. Была усовершенствована модель работы с компонентами (VCL), появилась возможность быстрого использования таких технологий, как MIDAS, DCOM и CORBA. В этой версии так же был усовершенствован язык Object Pascal: появились новые черты, свойственные современному ООП, были введены новые типы данных, включая 64-битные.

Delphi 5. Изменений в Delphi 4 было так много, что, к сожалению, не обошлось без досадных накладок и ошибок. Результатом был выпуск 3-х пакетов обновлений (наши пираты продавали их под вывесками Delphi 4.1, 4.2 и 4.3). Но самым полным и правильным пакетом обновлений следует считать Delphi 5 – в этой версии было доведено до ума все то, что было начато в Delphi 4.

Delphi 6. С выходом шестой версии Borland провозгласила новую, кросс-платформенную эпоху в Delphi, параллельно выпустив версию Delphi для ОС Linux (Kylix 1.0). Теперь при создании приложения можно было выбирать его тип – для Windows, или же универсальное. Соответственно, к библиотеке VCL добавилась библиотека CLX, совместимая как с Windows, так и с Linux. Кроме того, в 6-й версии наконец-то появились «родные», а не чужеродные ActiveX, компоненты для работы с Интернетом, причем сразу в обоих вариантах – и для VCL и для CLX. Начиная с этой версии, сам язык программирования Object Pascal официально называется Delphi.

Delphi 7. Дальнейшее развитие Delphi 6, улучшенная библиотека CLX и новая корреспондирующая версия среды разработки под ОС Linux – Kylix 3. Вместе с тем, начиная с Delphi 7, Borland озаботилась вопросом совместимости с еще одной платформой – Microsoft .NET, для чего некоторые изменения были внесены как в VCL, так и в язык (и, соответственно, в компилятор).

Delphi 8. Первая версия Delphi, ориентированная на работу с платформой Microsoft.NET. В этой версии разработчики рискнули изменить интерфейс среды, сделав его похожим на продукцию Microsoft.

Delphi 2005. Менее чем через год после выхода Delphi 8, появилась новая версия, в которой спешно вернули возможность работать в классическом стиле IDE при разработке приложений для Windows. Вместе с тем, Delphi 2005, в случае разработки приложений специально для платформы NET, позволяет работать не только с языком Delphi, но и с C#. При этом в сам язык Delphi были введены такие новшества, как оператор for…in и встраиваемые процедуры и функции.

Среда Delphi и простейшее приложение

Ознакомившись с теорией предмета изучения, перейдем к практическому его освоению. В этой главе вы узнаете, из чего состоит Delphi, какие программы с ее помощью можно делать, и, собственно, создадите первую программу в среде Delphi!

Состав Delphi и требования к системе

Прежде, чем приступать к работе с какой-либо программой, было бы полезно ознакомиться с ее требованиями к компьютеру. Разумеется, требования у разных версий Delphi разнятся, постепенно повышаясь от версии к версии. В частности, в рассматриваемой нами Delphi 7 рекомендуется процессор не ниже Pentium II и не менее 256 Мбайт оперативной памяти. Более ранние версии требовали меньший объем памяти, однако для комфортной работы я в любом случае рекомендовал бы не менее 256 Мбайт, а для Delphi 7 и выше, да еще и под управлением ОС Windows XP, не помешало бы иметь 512 Мбайт ОЗУ.

Что касается требований к операционной системе, то хотя формально Delphi может работать на любой 32-разрядной версии Windows, я бы настоятельно рекомендовал использовать Windows из линии NT, т.е. Windows 2000 или XP. Дело в том, что Windows 9x, из-за своего 16-разрядного наследия, имеет серьезные ограничения на количество доступных системных ресурсов, вне зависимости от того, насколько мощный ПК вы используете. Кроме того, Windows 9x не может эффективно задействовать даже относительно большие – свыше 128 Мбайт – объемы оперативной памяти. Я уже не говорю о том, что в Windows 9x не поддерживаются ни многопоточность, ни набирающие в последнее время популярность двуядерные процессоры, а производители аппаратных компонентов ПК давно уже забросили оптимизацию драйверов для данного семейства ОС. Результатом всего этого является низкая производительность на современных компьютерах и вполне ощутимый риск «повесить» систему в процессе работы над сложным и ресурсоемким приложением.

Еще один важный вопрос – это монитор. Опять-таки, формально достаточно любого SVGA-монитора. Но работать в среде Delphi при разрешении экрана ниже, чем 1024 на 768 точек, крайне затруднительно: учтите, что вам постоянно надо видеть как элементы управления самой Delphi, так и собственное (разрабатываемое) приложение. Для комфортной работы я бы рекомендовал качественный 19” монитор с рабочим разрешением 1280 на 1024 точки. Причем, если это будет обычный монитор на ЭЛТ (или даже ЖК, но с аналоговым подключением), то вам понадобится еще и качественная видеокарта, способная обеспечить кристально четкую, без «замыливания» картинку. Для ЭЛТ-мониторов также важно обеспечивать поддержку указанного разрешения при частоте регенерации изображения не ниже 85 Гц.

ПРИМЕЧАНИЕ

> Помните, что программирование – это напряженная работа с текстом. И если ваша связка «видеокарта-кабель-монитор» не может выдать четкий текст и (или) отсутствие видимого мерцания в нужном вам разрешении, то со временем вы рискуете испортить себе зрение.

Определившись с компьютером, перейдем к установке. В процессе установки программа спросит вас, для каких версий тех или иных третьесторонних приложений следует устанавливать компоненты. Прежде всего, это версии MS Office, для одной из них вы сможете установить набор компонент, обеспечивающих взаимодействие между приложениями office и Delphi. Если вы устанавливаете старшую версию Delphi (Client/Server, Enterprise, Architect), то вас спросят еще и о том, для каких версий баз данных следует установить компоненты. Наконец, в процессе установки, помимо самой Delphi будут установлено множество дополнительных программ, в основном,

связанных с базами данных. Причем некоторые из них (например, сервер InterBase или виртуальная Java-машина) вообще устанавливаются отдельно, хотя и в процессе общего хода инсталляции.

По завершению процесса установки в программном меню Windows будет сформирована группа Borland Delphi, в которой, помимо самой Delphi, будут находиться ярлыки всех вспомогательных компонент среды. В частности, там будут находиться ярлыки для следующих программ:

- Image editor – простой графический редактор для рисования иконок и курсоров. За время, прошедшее с момента последнего обновления (в 1996 году), морально устарел, но может пригодиться, если нет ничего другого;
- WinSight – позволяет просматривать отладочную информацию в любых работающих приложениях;
- BDE Administrator – позволяет настраивать базы данных;
- Data Pump – позволяет переносить данных между БД;
- Database Explorer или SQL Explorer – средство просмотра БД;
- SQL Monitor (только старшие версии) – позволяет отслеживать обращения приложений к SQL-серверу.

Кроме того, в этой группе будет находиться подгруппа Help, а в ней, среди множества справочных файлов, – еще одна, с еще большим их количеством – MS SDK Help Files. Так вот, все эти файлы вам придется регулярно использовать, причем положение усугубляется не только их количеством и объемами, но и тем, что в русском варианте их не существует. Таким образом, знание английского языка будет вам хорошим подспорьем при изучении как Delphi, так и программирования вообще.

СОВЕТ

> Если вы не знаете английского, но владеете немецким или французским, то вы можете установить версию Delphi на том языке, который знаете лучше. Русских версий Delphi, равно как и других серьезных средств разработки, нет, никогда не было, и, увы, даже не предвидится.

Таким образом, мы вкратце ознакомились с составом среды Delphi. Отдельные вспомогательные приложения, при необходимости, мы еще рассмотрим, ну а сейчас пора приступать к изучению главной составляющей – самой среды Delphi.

Интегрированная среда разработки

Интегрированная среда разработки Delphi (Delphi IDE) является многооконной системой. Она включает в себя все необходимое для быстрой разработки Windows-приложений, и может гибко настраиваться.

Тем не менее, как и всякая другая программа, Delphi имеет некоторый стандартный, предусмотренный разработчиками вид, в котором она предстает вам при первом запуске. В таком «стандартном» варианте среда Delphi имеет 6 окон (рис. 2.1). Это: главное окно (Delphi 7 – Project1), окно дерева объектов (Object TreeView), окно инспектора объектов (Object Inspector), окно конструктора форм (Form1), а так же со-

вмещенное окно редактора кода и проводника кода (на заднем плане, под Form1). При этом окно проводника пристыковано к левому краю окна редактора. Впрочем, ничего не мешает отсоединить проводник от редактора, или, наоборот, состыковать все окна, кроме главного и конструктора форм, в одном окне, или объединить их по какому-либо иному принципу.

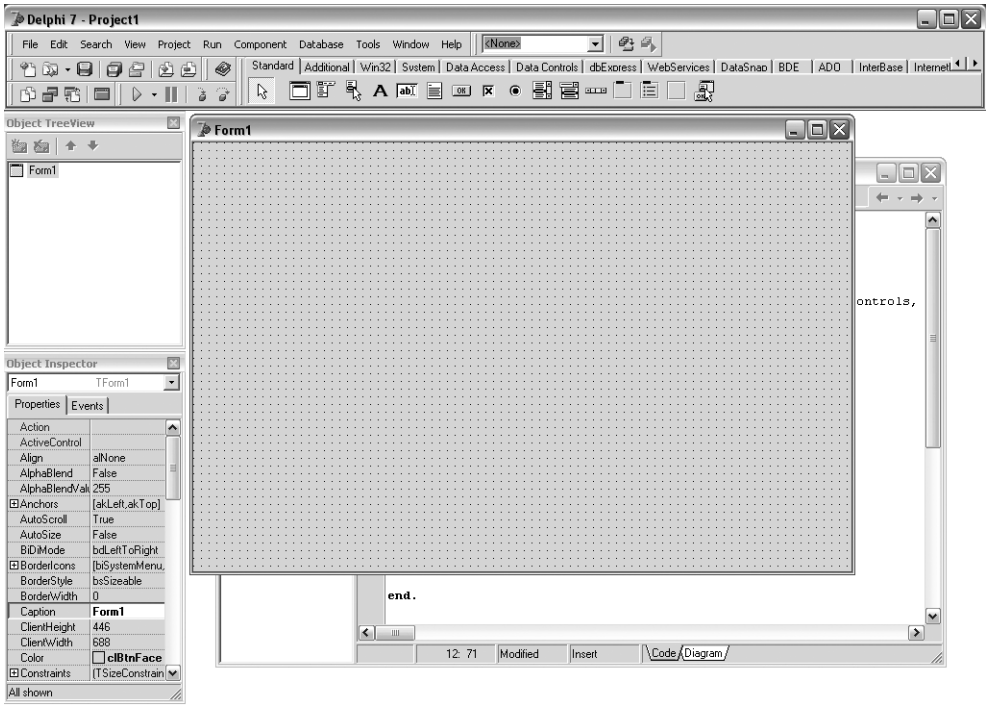

Рис. 2.1. Вид Delphi 7 IDE по умолчанию

К вопросу об удобстве следует отметить, что предлагаемая разработчиками компоновка годится, в принципе, для любого экранного разрешения. Но если у вас имеется возможность установить разрешение экрана в значение 1280 на 1024 точки, то вы можете скомпоновать все кнопки главного окна в одной строке, а все освободившееся внизу место выделить для палитры компонентов (рис. 2.2).

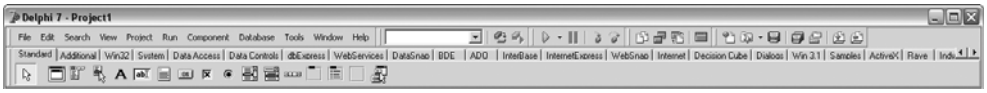

Рис. 2.2. «Оптимизированный» вид главного окна

Такой шаг позволит иметь перед глазами гораздо большее количество категорий компонент, что положительно сказывается на продуктивности работы. В принципе, это можно сделать и при более низком разрешении, однако при этом частью кнопок на панелях инструментов придется пожертвовать.

Содержание и предназначение панелей инструментов, имеющихся в начальном виде, следующее:

- Debug – отладка. Позволяет запустить программу (Run), приостановить ее выполнение (Pause), а так же выполнять построчное выполнение программы;
- Standard – стандартные. Служит для таких операций, как сохранение, создание, добавление и удаление файлов;
- View – вид. Используется для быстрого нахождения форм и файлов проекта;
- Desktops – рабочая среда. С помощью этих инструментов можно переключаться между различными настройками рабочей среды Delphi;
- Custom – произвольная. Изначально содержит одну-единственную кнопку – для вызова справки;
- Component palette – палитра компонентов. Содержит все доступные для разработки приложений компоненты.

Отметим, что все инструментальные панели настраиваются: кнопки можно перемещать между панелями, добавлять на них новые, или же удалять. Для обычных панелей (Standard, View, Debug) это делается точно таким же образом, как во многих других современных Windows-приложениях (например, как в Word, т.е. при помощи окна настройки – Customize).

Что касается самой большой панели – палитры компонентов, то для ее настройки следует использовать специальное окном свойств палитры (рис. 2.3). Это окно доступно через пункт Configure Palette из меню Component. Однако учтите, что при настройке важно знать как предназначение компонент, так и понимать принципы их организации, поэтому максимум, что можно себе позволить для начала – это поменять местами группы, перетаскивая их в списке страниц (Pages).

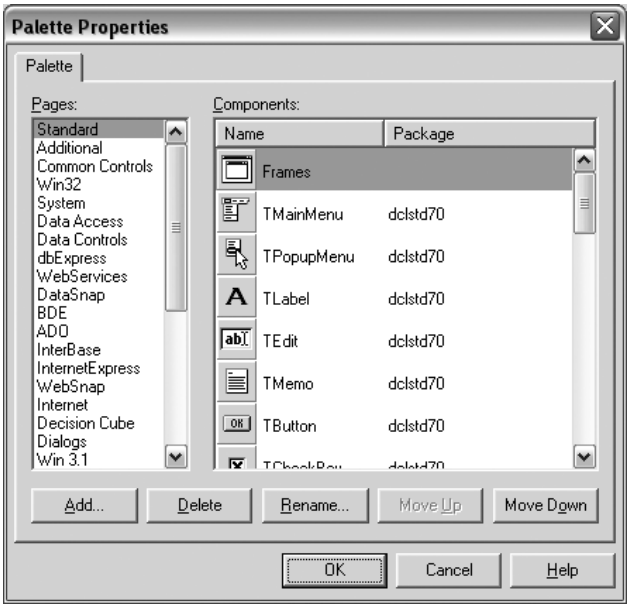

Рис. 2.3. Настройка палитры компонентов требует знания VCL

ПРИМЕЧАНИЕ

Следует учитывать, что поскольку палитра компонентов является ничем иным, как визуальным представлением VCL, то ее вид и состав могут меняться в зависимости от того, какие модули подключены, устанавливались или нет дополнительные компоненты или их наборы, и т.д. В любом случае, в начале изучения Delphi, экспериментов в этой области лучше не производить.

Все компоненты сгруппированы по вкладкам, число и состав которых несколько разнятся в зависимости от версии и варианта поставки. Так, в Delphi 7 Enterprise имеется 33 вкладки, содержащие компоненты, принадлежащие к той или иной группе VCL (табл. 2.1).

Таблица 2.1. Страницы палитры компонент Delphi 7 Enterprise

Страница	Название	Описание
Standard	Стандартные	Основные элементы интерфейса приложений Windows (меню, кнопки, подписи и т.п.)
Additional	Дополнительные	Набор улучшенных элементов управления, имеющихся в VCL
Win32	32-разрядные Windows	Элементы интерфейса приложений, характерные для Windows 95 и последующих версий этой ОС
System	Системные	Элементы управления и доступа к системным функциям Windows (таймер, OLE, DDE)
Data Access	Доступ к данным	Стандартный набор компонент для доступа к БД
Data Controls	Элементы данных	Элементы пользовательского интерфейса для доступа к БД
dbExpress	dbExpress	Компоненты для доступа к БД при помощи SQL-драйвера dbExpres
WebServices	WebServices	Компоненты для взаимодействия с удаленный web-сервером через SOAP
DataSnap	DataSnap	Компоненты для взаимодействия с сервером через DCOM
BDE	Borland Database Engine	Компоненты для доступа к БД посредством BDE (классический вариант для простых БД)
ADO	ADO	Компоненты для взаимодействия с БД через ADO
InterBase	InterBase	Компоненты для прямого взаимодействия с БД InterBase
InterBase Admin	Администрирование InterBase	Компоненты для взаимодействия и управления сервером БД InterBase
InternetExpress	InternetExpress	Компоненты для взаимодействия с данными через XML
WebSnap	WebSnap	Компоненты для работы с данными через различные протоколы Интернета
Internet	Internet	Набор ActiveX-компонент для работы через Интернет
Decision Cube	Decision Cube	Набор компонент для обработки информации в БД
Dialogs	Диалоги	Стандартные и расширенные диалоговые окна
Win 3.1	Windows 3.1	Компоненты пользовательского интерфейса, харак-

		терные для Windows 3.1
Samples	Примеры	Несколько визуальных компонент, не являющихся официально поддерживаемыми
ActiveX	ActiveX	Несколько встраиваемых ActiveX-приложений
Rave	Rave Reports	Набор компонент для построения отчетов
Indy Clients	Клиенты Indy	Набор компонент-клиентов для различных протоколов и служб Интернета
Indy Servers	Серверы Indy	Набор компонент-серверов для различных протоколов и служб Интернета
Indy Intercepts	Обработчики Indy	Набор компонент, позволяющих отлавливать сообщения от клиентов и серверов Indy
Indy i/o handlers	Ввод-вывод Indy	Компоненты для отслеживания активности соединений других компонент Indy
Indy Misc	Утилиты Indy	Набор вспомогательных компонент, полезных при разработке различных TCP-приложений
COM+	COM+	Содержит компонент, позволяющий создать управляющий сервер COM+
IW Standard, Data, Client Side, Control	IntraWeb	Набор специальных кросс платформенных компонент для создания Web-приложений для любых Web-клиентов, включая КПК и смартфоны
Servers	MS Office Servers	Набор ActiveX-компонент для взаимодействия с приложениями Microsoft Office

Суммарно Delphi включает в себя сотни компонент, однако не стоит переживать по поводу огромного их количества: Delphi применяется во многих областях, и вряд ли хоть один разработчик в действительности использовал все доступные компоненты. Так что мы выделим наиболее полезные для нас группы, а именно: стандартные, дополнительные, 32-разрядные Windows, системные и диалоги. Этого набора будет более чем достаточно для начала изучения Delphi. Со временем мы ознакомимся так же с классическими компонентами для доступа к БД (Data Access и Data Controls), а так же с несколькими компонентами из богатой коллекции Indy. На этом введение в палитру компонент можно считать завершенным и перейти к дальнейшему ознакомления со средой, для чего перейдем к детальному исследованию главного окна, начав с его меню, которое состоит из 11 пунктов:

- File – файл. Операции с файлами, вроде создать, открыть, сохранить;
- Edit – правка. Операции редактирования, как стандартные для текстового процессора (отмена, копирование-вставка), так и специфические для редактирования разрабатываемых окон приложений (выравнивание, порядок создания и т.п.);
- Search – поиск. Различные варианты поиска и замены;
- View – вид. Переключение между различными окнами – как относящимися к IDE, так и к разрабатываемому приложению;
- Project – проект. Все операции по работе с проектом, как то добавление и удаление файлов, настройки, сборка и компиляция;
- Run – выполнить. Средства для отладки программ;

- Component – компоненты. Средства для работы с компонентами, включая настройку палитры компонент;
- Database – Данные. Некоторые средства для работы с БД;
- Tools – сервис. Настройка параметров IDE, а так же вызов вспомогательных программ (Image editor и др.);
- Windows – окно. Содержит список всех открытых в текущий момент окон и позволяет переключаться между ними (актуально, когда окон много и одни загораживают другие);
- Help – справка.

Таким образом, мы уже исследовали все главное окно IDE Delphi. Что касается остальных окон, то их предназначение мы узнаем по ходу дальнейшего ознакомления со средой, а относительно их расположения, да и вообще необходимости в некоторых из них, то это – вопрос личных вкусов и специфики разрабатываемого приложения – общие рекомендации тут были бы неуместны.

Проекты в Delphi

Разработка приложений в Delphi означает работу с проектами. Иначе говоря, когда вы приступаете к разработке собственной программы в Delphi, первым делом создается проект – группа файлов, представляющих исходные данные (прежде всего, код) для приложения. Одни из этих файлов создаются во время разработки приложения (собственно программный код, включая файл проекта, и представленные в виде кода формы), другие же создаются автоматически при запуске программы. Таким образом, все файлы проекта подразделяются на следующие типы:

- dpr – собственно файл проекта;
- pas – модули приложения, содержащие код на Object Pascal;
- dfm – модули приложения, содержащие информацию об окнах приложения;
- res – файлы со встраиваемыми ресурсами приложения (например, иконками);
- obj – файлы, содержащие готовый к компиляции объектный код;
- cfg, dof, dsk – служебные файлы Delphi.

Основными составными частями проекта, помимо самого файла проекта (dpr), являются модули pas и dfm. При этом для каждого модуля окна (dfm) имеется собственный программный модуль (pas).

Чтобы лучше во всем этом разобраться, попробуем создать собственный проект в Delphi. Для этого достаточно запустить Delphi – в том случае, если вы не изменяли настроек, новый проект создастся автоматически. Но на всякий случай, мы все-таки создадим новый проект самостоятельно, для чего следует из меню File зайти в группу New и выбрать в ней пункт Application.

ВНИМАНИЕ

> Чтобы постоянно не повторяться, в дальнейшем подобные последовательности действий мы будем обозначать следующим образом: File → New → Application.

В результате мы получим новый проект, полностью готовый к дальнейшему использованию (см. рис. 2.1). Более того, его уже на этом этапе можно выполнить! Для этого достаточно нажать на кнопку Run, находящуюся на панели инструментов отладки, или же выбрать пункт Run из одноименного меню (Run → Run), но лучше всего нажать на клавишу F9: для быстрой разработки приложений в среде Delphi знание хотя бы основных сочетаний «горячих клавиш» просто необходимо.

Итак, мы запустили приложение, правда, выглядеть оно будет довольно скучно: пустое окно с заголовком Form1 и стандартными кнопками управления окном (рис. 2.4). Но, по крайней мере, даже такое приложение обладает всей базовой функциональностью: его можно развернуть на весь экран, или наоборот, свернуть в панель задач, перемещать по экрану, изменять размеры, и, что самое главное – закрыть.

Рис. 2.4. Самое первое приложение

Теперь немного модернизируем свое приложение, заодно изучив еще одно важное окно среды Delphi – Object Inspector. Для этого вернитесь в рабочую среду Delphi, закрыв запущенное приложение, и щелкните по окну Form1, чтобы оно стало активным. Это окно, как и любые другие окна, относящиеся непосредственно к разрабатываемому приложению, называют формой. Теперь обратите внимание на окно инспектора объекта (рис. 2.5), по умолчанию оно расположено по левому краю экрана.

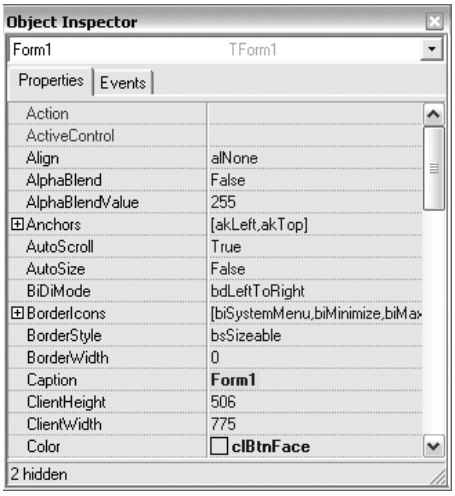

Рис. 2.5. Окно инспектора объектов

В его верхней части находится ниспадающий список, содержащий все элементы, интерфейса выбранной формы, в данном случае там будет только сама форма, отмеченная в виде имени объекта (Form1). Далее располагается список всех свойств объекта, доступных для изменения в процессе визуальной разработки.

ПРИМЕЧАНИЕ

> Все свойства объектов в Delphi, доступные для изменения, делятся на run-time (времени выполнения) и на design-time (времени разработки). При этом первые можно изменять только во время работы программы, а вторые доступны уже во время визуального редактирования.

Попробуем заменить значение одного из свойств. Наиболее безопасным будет изменение такого свойства, как Caption – оно отвечает за текст, находящийся в заголовке окна. Для изменения значения этого свойства, щелкните по строке с ним и вместо «Form1» введите какой-либо собственный текст, например, «Мое первое приложение в Delphi». В данном случае вы сразу же увидите результат своей работы: заголовок окна формы изменится на новый.

Некоторые свойства меняются не путем непосредственного ввода значений, а путем выбора одного из заранее предусмотренных. В простейшем случае это может быть выбор ложь-истина (False или True), включающим или отключающим ту или иную опцию. Иногда списки бывают гораздо более объемными. Например, для выбора цвета предлагается множество цветов, включая системные. Например, свойство Color имеет в нашем случае значение clBtnFace, что означает цвет кнопки, установленный в настройках Windows. Мы можем изменить его на любой другой цвет, как системный (например, clCaptionText – цвет текста заголовка окна), так и на явный, например, clWhite (белый).

Можно заметить, что некоторые из свойств отмечены значком «+». Это означает, что такое свойство является составным, и если щелкнуть по значку, то раскроются строки, содержащие отдельные параметры. Например, можно раскрыть таким образом свойство BorderIcons и поменять значение параметра biMinimize с True на False. Тем самым мы изменим значение свойства, отвечающего за системные элементы управления окна таким образом, что функция разворачивания окна на весь экран будет недоступной.

Есть и такие свойства, для редактирования которых открывается отдельное окно. Например, тот же цвет можно определить не выбирая его из списка предусмотренных вариантов, а открыв стандартное для Windows окно выбора цвета. Для этого достаточно сделать двойной щелчок мышкой по области списка цвета. В других случаях (например, для выбора свойств шрифта – Font) окно настроек можно вызвать, нажав на имеющуюся напротив таких свойств кнопку с многоточием.

Ну а пока что сведем воедино все измененные нами свойства формы Form1:

```
Caption: Мое первое приложение в Delphi
Color: clWhite
BorderIcons: [biSystemMenu,biMinimize]
```

Последнее свойство – составное и такое его значение получается в результате установки таких его составляющих, как biSystemMenu и biMinimize в True, а biMaximize и biHelp – в False.

Часть из сделанных изменений – цвет окна и его заголовок – видна сразу же, в рабочей среде Delphi, т.е. на этапе разработки. А вот изменения в системных кнопках, хотя и могут быть сделаны на данном этапе, визуально себя не проявляют. Поэтому, чтобы увидеть сразу все произошедшие изменения, запустим приложение на выполнение, нажав клавишу F9, и вы увидите, что не только цвет и заголовок окна изменились, но и кнопка «развернуть» стала неактивной (рис. 2.6).

Рис. 2.6. Первое приложение после небольшой доработки

Таким образом, мы ознакомились с Object Inspector – одним из наиболее важных окон рабочей среды Delphi. Ну а чтобы завершить тему введения в проекты, попробуем сохранить наш проект на диске. Пусть это будет папка Project1, а сам проект мы назовем first. Для этого откройте диалог сохранения проекта (File → Save project as) и выберите в нем нужную папку.

СОВЕТ

> По умолчанию Delphi предлагает складывать все проекты в недра своей собственно директории в Program Files. Но было бы гораздо лучше, если бы вы создали папку в другом, легко доступном месте, и назвали ее понятным для себя названием. Например, это может быть папка Work на диске C:.

А теперь внимание! Если вы сохраняете проект в первый раз, то Delphi предложит вам сначала сохранить не файл проекта, а все несохраненные рабочие файлы. В данном случае это будет программный файл формы. По умолчанию Delphi предложит назвать его unit1.pas, но лучше сразу взять за правило давать осмысленные имена все рабочим файлам. В частности, раз это окно – главное (и единственное) в нашем приложении, то назовем его файл main.pas. Таким образом будут сохранено сразу 2 файла – программный pas и файл формы dfm.

ВНИМАНИЕ

> Имена любых файлов проекта должны состоять только из латинских букв и цифр, при этом начинаться должны с буквы. Также в них недопустимы пробелы и любые специсмволы, кроме знака подчеркивания.

Только после сохранения всех составных частей, будет предложено сохранить собственно файл проекта. Назовем его «first». После сохранения можно, наконец-то скомпилировать исполняемый файл, нажав Ctrl+F9 (или Project → Make), закрыть Delphi и посмотреть, что мы имеем в папке Project1. А в ней, как и было обещано, будет

файл main.pas – программный код для формы, main.dfm – описание формы, first.dpr – сам проект, first.res – файл ресурсов проекта, main.dcu – подготовленный к компиляции модуль, и, разумеется, first.exe – исполняемый файл готового приложения. Так же вы обнаружите в нем все служебные файлы Delphi, в которых хранится дополнительная информация о проекте и настройках рабочей среды для него – файлы first.cfg, first.dof и first.dsk.

Чтобы теперь вернуться к работе над этим проектом, достаточно дважды щелкнуть по файлу first.dpr, в результате будет загружена и Delphi IDE, и этот проект в нее.

Типы проектов и депозитарий

Только что мы рассмотрели создание наиболее распространенного типа проекта – приложения Windows со стандартным графическим интерфейсом. Но на самом деле, возможности Delphi этим не ограничены, вы можете создавать приложения самого разнообразного характера, включая консольные (для текстового режима Windows), динамически подключаемые библиотеки (DLL), сервисы для Windows NT/2000/XP, межплатформенные приложения CLX (Delphi 6,7) или приложения для платформы Microsoft .NET (Delphi 8, 2005). Чтобы создать приложение определенного типа, следует из подменю File → New выбрать пункт Other. Таким образом, откроется окно, позволяющее выбрать тип нового приложения или добавить какой-либо специфический модуль к существующему проекту (рис. 2.7).

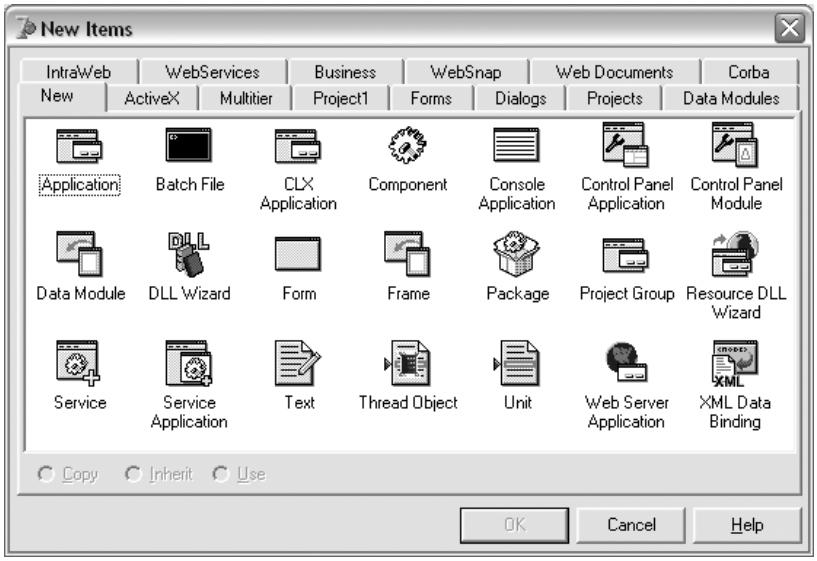

Рис. 2.7. Выбор нового приложения или модуля в Delphi 7

Выбор вариантов тут весьма обширен, причем, помимо типовых модулей и классов приложений, присутствуют различные мастера, позволяющие упростить процесс создания того или иного модуля (Wizard's), а так же специализированные стандартные формы вроде диалоговых окон или окна «О программе». Рассмотрим некоторые из них подробнее, для чего пройдемся по отдельным закладкам окна New Items.

Начнем с закладки New. На ней представлены наиболее часто востребованные, по мнению разработчиков, варианты. И действительно, тут можно найти стандартное графическое Windows-приложение (Application), форму (Form), программный модуль (Unit), текстовое приложение командной строки (Console Application), и другие варианты, как-то Data Module (полезен для разработки баз данных), DLL Wizard, Component и т.д.

На закладках Forms и Dialogs моно найти ряд стандартных диалоговых окон и даже мастер по разработке диалогового окна.

Закладка Projects дает возможность начать проект того или иного типа, или даже воспользоваться мастером для создания многооконного приложения.

Чтобы создать элемент управления ActiveX или приложение для COM+, следует обратиться к шаблонам на закладке ActiveX. Ну а прочие закладки, в том числе IntraWeb, WebSnap и т.д., позволяют создавать специализированные приложения соответствующего типа или модули к ним. Их количество и названия зависят от версии Delphi и варианта поставки.

Но на самом деле, данное окно, по большому счету, подобно палитре компонент, с тем лишь исключением, что если палитра компонент являет собой представление VCL, то окно New Items – во многом является отображением депозитария (Object Repository). В депозитарии хранятся заготовки форм и иных модулей, которые вы можете многократно использовать в своих проектах. При этом для того, чтобы поместить форму в депозитарий, достаточно воспользоваться ее контекстным меню и выбрать в нем пункт Add to Repository.

Прочие средства IDE

На текущий момент мы уже рассмотрели такие составные части, предоставляемые интегрированной средой разработки Delphi, как главное окно вместе с его меню, окном выбора модулей и палитрой компонентов, и инспектора объектов. Теперь обратимся к такой важной части, как окно редактора кода. Следует отметить, что до появления графических средств разработки, подобных Delphi, еще во времена MS-DOS и ранних версий Windows, IDE для программирования как раз и состояли из редактора кода да самого компилятора. Таким образом, редактор кода – это наиболее характерный и устоявшийся элемент в любой среде разработки приложений.

Применительно ко всем современным версиям Delphi, редактор кода имеет все стандартные возможности редактирования текста (вроде работы с буфером обмена), а так же ряд особенностей, характерных для редакторов кода, а именно:

1. Редактор всегда работает с моноширинным шрифтом (т.е. все буквы имеют одинаковую ширину). Это необходимо, поскольку в противном случае было бы тяжело ориентироваться в коде программы;
2. Моноширинный режим позволяет использовать такой метод, как колоночную разбивку. Иначе говоря, копировать и перемещать можно не только отдельные слова или строки, но и вырезать, копировать и вставлять прямоугольные фрагменты текста;
3. Редактор постоянно отображает позицию курсора, т.е. в какой строке и колонке находится точка ввода;

4. Отсутствие автоматического переноса строк. Поскольку в программировании каждый символ, включая обрыв строки, что-то значит, то чтобы программисты не гадали, где в коде конец строки, а где автоматический перенос, такого режима правки нет в принципе. К тому же это мешало бы ориентироваться в номерах строк;
5. Подсветка синтаксиса выделяет ключевые слова и прочие специфические языковые конструкции;
6. При перемещении по тексту стрелкой вправо по окончании текста в строке, курсор не переносится на сроку вниз, а продолжает перемещаться дальше;
7. Вы можете устанавливать закладки, т.е. помечать место в тексте при помощи сочетания горячих клавиш и быстро перемещаться между ними;
8. Также предусмотрены такие функции, как автоподстановка кода по ключевым фразам, автозавершение кода и т.д.

При всем этом следует отметить, что практически все функции редактора, включая правила подсветки, сочетания горячих клавиш, поведение курсора, автоподстановка и т.д., гибко настраиваются. Для этого следует открыть окно свойств редактора (Tools → Editor options) и настроить параметры на свое усмотрение. Впрочем, если у вас еще не сложились какие-то предпочтения в работе с подобными редакторами, то можно этого и не делать – предлагаемые изначально настройки вполне удобны.

Что касается внешнего вида окна редактора, то в нем, при стандартных параметрах IDE, помимо области редактора кода имеется еще и проводник кода, упрощающего процесс навигации по файлу (рис. 2.8).

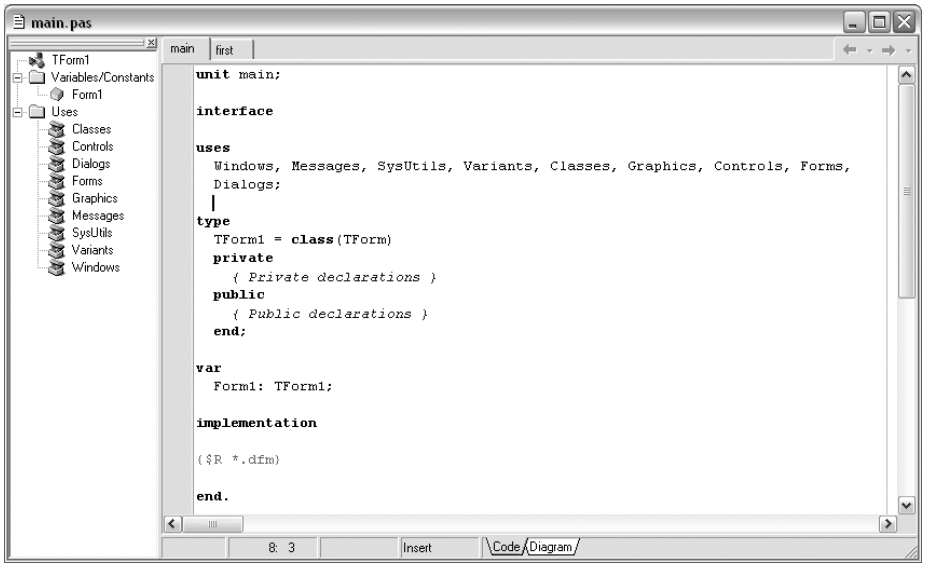

Рис. 2.8. Окно редактора кода с проводником и загруженным файлом новой формы

Обратите внимание на то, что весь программный код файла main.pas, показанный в этом окне, был создан автоматически, равно как и код для файла проекта – first.dpr.

Иначе говоря, всю работу по созданию базовых блоков приложения интегрированная среда Delphi делает за вас.

Завершая тему окна редактора, отметим, что для того, чтобы загрузить в него какой-либо произвольный файл, следует воспользоваться главным меню (File → Open). При открытии проекта файлы открытых форм загружаются в него автоматически, а чтобы загрузить в него иные файл проекта, используйте кнопку View Unit (сочетание горячих клавиш – Ctrl+F12) на инструментальной панели View. Если же требуется загрузить как исходный код, так и саму форму, то воспользуйтесь находящейся по соседству кнопкой View Form (Shift+F12).

Ну и последнее не рассмотренное нами окно – Object TreeView, или окно дерева объектов, служит для просмотра иерархии элементов управления (кнопок, переключателей и т.п.) относительно формы. Т.е., если проводник кода упрощает процесс поиска того или иного компонента в исходном коде программы, то дерево объектов поможет быстрее сориентироваться в элементах, находящихся на форме.

Создание приложения командной строки

Итак, мы уже ознакомились со всеми основными функциями IDE Delphi, и даже создали простейшее приложение для Windows. Однако целью данной части книги является все-таки изучения основы основ – языка Object Pascal. Поэтому для того, чтобы не отвлекаться по ходу его изучения на второстепенные (по отношению к самим языковым конструкциям) детали, рассмотрим вариант создания консольного приложения, т.е. фактически, программы для DOS. Не стоит думать, что это будет шагом назад, или что это давно морально устарело. На самом деле, абсолютно все правила языка действуют совершенно одинаково для любых программ, будь они под DOS, Windows, .NET, или Linux. Вместе с тем, отсутствие необходимости в параллельном изучении специфических для конкретной платформы особенностей (а уж тем более – в параллельном изучении такой обширной библиотеки, как VCL!) существенно упростит нашу задачу. Более того, консольные приложения в типичном случае могут состоять всего лишь из одного единственного файла, что так же упрощает понимание предмета изучения, коим на настоящий момент является сам язык программирования. Ну а после того, как будет выучен язык, применить его для создания полноценных Windows-приложений не составит труда, да и изучение VCL станет легче ввиду того, что будет ясно, что, как и почему в ней устроено.

После такого небольшого отступления, приступим к созданию первого консольного приложения в среде Delphi. Для этого откройте окно New Items (File → New → Other) и на закладке New дважды щелкните по значку Console Application. В результате откроется окно редактора с загруженным в него проектом (листинг 2.1).

Листинг 2.1. Заготовка приложения командной строки

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

Первой строкой идет название программы, в данном случае это Project1, затем Delphi вставило для себя указание, что это – приложение для командной строки, после чего следует ключевое слово «uses» и перечисление необходимых дополнительных файлов (sysutils), ну и после этого, со слова begin начинается собственно тело программы. Завершается любая программа в Pascal ключевым словом end с точкой. Между ключевыми словами begin и end, в фигурных скобках, вставлен автоматический комментарий, не влияющий на выполнение программы, так что при желании можно его удалить.

А теперь самостоятельно напишем первую программу в Delphi! По традиции, она у нас будет выводить фразу «Hello, World!». Для этого на том месте, где находился комментарий, напишем одну строчку кода:

```
write('Hello, World!');
```

Все! Теперь можно сохранить и скомпилировать нашу программу. Для этого щелкните по кнопке Save или Save All на стандартной панели инструментов, в качестве пути к файлу укажите какой-либо каталог на вашем жестком диске (например, создайте папку HelloWorld на диске C:), а саму программу можно назвать hello. Таким образом, все наше приложение будет сохранено, а исходный код примет вид, показанный в листинге 2.2.

Листинг 2.2. Программа hello

```
program hello;
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
  write('Hello, World!');
end.
```

Обратите внимание, что название изменилось автоматически. Теперь остается получить исполняемый (exe) файл, для чего скомпилируем программу, нажав Ctrl+F9. Теперь запустим программу. Поскольку она у нас рассчитана на режим командной строки, то для начала откроем командную строку (Пуск → Все программы → Стандартные → Командная строка в Windows XP, или Пуск → Программы → Стандартные → Сеанс MS-DOS в Windows 98). В командной строке вызовем нашу программу, не забыв указать полный путь к ней. Например, если вы сохранили проект в C:\HelloWorld, а саму программу назвали Hello, то и путь укажите соответствующий, т.е.:

```
C:\HelloWorld\hello.exe
```

Запустив программу (т.е. введя в командной строке путь и нажав Enter), вы сразу же увидите результат ее выполнения – она выведет фразу «Hello, World!», и завершится. Собственно говоря, именно по этой причине мы сначала открыли командную строку, а лишь затем запустили программу, поскольку непосредственный ее запуск (через Проводник Windows, или прямо из Delphi – по F9), привел бы к тому, что на экране просто мелькнул бы автоматически запущенный сеанс MS-DOS и сразу закрылся бы. Это объясняется тем, что программа завершает свою работу сразу после того, как будет достигнут ее конец, обозначенный как «end.», а это в нашем случае произойдет моментально после вывода текста. Но мы можем изменить такое ее поведение, добавив еще одну сроку кода непосредственно после «write('Hello, World!');», которая

будет дожидаться момента, пока пользователь не нажмет клавишу Enter. Выглядеть она будет так:

```
readln;
```

Теперь можно запустить наше приложение прямо из среды Delphi, нажав F9. В результате можно будет наблюдать надпись «Hello, World!» на фоне черного окна автоматически запущенного сеанса консоли командной строки до тех пор, пока вы не нажмете Enter.

Исходный код программы вы найдете в папке Demo\Part1\HelloWorld на прилагаемом компакт-диске.

Введение в Object Pascal

На текущий момент мы уже умеем создавать приложения Delphi. Тем самым все готово для того, чтобы приступать к изучению языка Object Pascal. Прежде всего, мы ознакомимся с синтаксисом языка – его алфавитом, структурой, ключевыми словами, а так же с правилами составления выражений и типами данных.

Алфавит и словарь языка

В программах Object Pascal могут использоваться любые символы из множества символов языка Object Pascal. К этому множеству относятся буквы латинского алфавита, арабские цифры, пробельные символы, разделители и специальные символы.

Прописные и строчные латинские буквы, а так же знак подчеркивания:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l
m n o p q r s t u v w x y z _
```

Десятичные цифры:

```
0 1 2 3 4 5 6 7 8 9
```

Пробельные символы (разделители):

```
Пробел, табуляция, перевод строки, возврат каретки
```

Специальные символы:

```
. , ; : ‘ ! | / ( ) { } < > [ ] # $ @ ^ - = + * " '
```

Составные символы, образуемые сочетанием двух специальных:

```
:= <> .. <= >= (* *) (. .) //
```

ПРИМЕЧАНИЕ

> Все прочие символы, включая символы кириллицы, также могут использоваться в Object Pascal, но только внутри строковых переменных и комментариев.

Последовательности, составленные из знаков алфавита, образуют слова. Слова отделяются друг от друга разделителями и несут в программе определенный смысл. В качестве разделителей могут использоваться как пробельные символы, так и некоторые специальные символы (например, ;), и комментарии. Сами слова можно разделить на такие группы, как ключевые слова; стандартные идентификаторы и пользовательские идентификаторы.

Что касается комментариев, то никакого смысла в программе они не несут, и могут использоваться для того, чтобы разработчик мог вставить пояснительный текст в код программы. Комментарии бывают двух видов – однострочные и многострочные. Для однострочных комментариев обычно используют составной символ «//», помещая его перед текстом самого комментария:

```
// эта строка полностью закомментирована
x=5; // эта часть строки - комментарий
```

Для многострочных комментариев применяют символы { и }, либо (* и *):

```
{ эти
строки
закомментированы }
(* и эти –
тоже! *)
```

Комментарии разных типов можно вкладывать друг в друга:

```
{ начало общего комментария
 (* этот комментарий был тут раньше *)
 // и этот – тоже
 а это – последняя строка общего комментария
}
```

С помощью комментариев удобно исключать отдельные инструкции или целые блоки программы в процессе ее отладки.

Ключевые слова

Ключевые, слова являются неотъемлемой частью языка. Все они имеют однозначно определенный смысл, изменить который невозможно. Кроме собственно ключевых слов бывают еще и зарезервированные, т.е. те, которые могут стать ключевыми в последующих версиях языка. Полный список ключевых слов Delphi приведен в таблице 3.1.

Таблица 3.1. Зарезервированные и ключевые слова в Delphi

Слово	**Слово**	**Слово**	**Слово**
absolute	export	nil	requires
abstract	exports	nodefault	resident
and	external	not	resourcestring
array	far	object	safecall
as	file	of	set
asm	finalization	on	shl
assembler	finally	or	shr
at	for	out	stdcall
automated	forward	overload	stored
begin	function	override	string
case	goto	package	then
cdecl	if	packed	threadvar

class	implementation	pascal	to
const	implements	private	try
constructor	in	procedure	type
contains	index	program	unit
default	inherited	property	until
destructor	initialization	protected	uses
dispid	inline	public	var
dispinterface	interface	published	virtual
div	is	raise	while
do	label	read	with
downto	library	readonly	write
dynamic	message	record	writeonly
else	mod	register	xor
end	name	reintroduce	
except	near	repeat	

Важно запомнить, что данные слова можно использовать только по их прямому назначению. Соответственно, вы не можете создавать какие-либо собственные идентификаторы (например, названия переменных), таким образом, чтобы если они совпадали с каким-либо ключевым словом.

Переменные и константы

Если ключевые слова имеют заранее предопределенный смысл, то значения констант и переменных определяет программист. Но прежде, чем приступить к исследованию констант и переменных, рассмотрим знак, используемый для некоторых операций, поскольку без этого невозможно дальнейшее продвижение. Речь идет о знаке равенства (=), который используется в Delphi в двух значениях – как оператор присваивания и как оператор сравнения.

В случае применения в качестве оператора присваивания, знак равенства используется совместно с символом двоеточия и имеет следующий общий синтаксис:

```
переменная := значение
```

При присваивании используется 2 операнда, при этом подразумевается, что левому операнду (переменной) будет присвоено значение правого. Такая запись называется выражением.

ПРИМЕЧАНИЕ

> Более точно выражение можно определить следующим образом: выражение – это набор данных, переменных, операторов и других выражений, которые приводятся к общему значению.

Другой вариант – это использование знака равенства для сравнения 2 операндов. В таком случае он используется самостоятельно:

```
операнд1 = операнд2
```

Такая запись означает, что сравниваются левое (операнд1) и правое (операнд2) значения выражения. Однако специально для констант предусмотрен третий вариант, при котором константе присваивается значение правого операнда (как в случае обычного присваивания), но используется знак равенства без двоеточия:

```
константа = значение
```

Таким образом, мы переходим к самим константам и переменным, и начнем мы с констант, как наиболее типичного примера подстановки.

Суть использования констант состоит в том, что вместо какого-либо явного значения указывается его условное обозначение – константа. Допустим, что вы пишете программу, в которой неоднократно следует вычислять НДС. Разумеется, вы можете использовать в выражениях явное значение – 0.18 (18%). Но скорее всего, в программе найдется несколько мест, где в вычислениях требуется значение НДС. Таким образом, если НДС в очередной раз изменят, то вам придется отыскивать в программе все эти строки и вносить правку. В таких случаях на помощь приходят константы – достаточно один раз ее определить, а затем во всех тех местах, где требуется ее значение, указывать имя константы.

Определяются константы при помощи ключевого слова const (от англ. constant):

```
const
  VAT = 0.18;
```

Теперь во всех выражениях, где требуется значение НДС, просто используется эта константа:

```
VATsumm := price * VAT;
```

В этом выражении задействована константа VAT и 2 переменных – VATsumm, которой присваивается значение, и price, которая используется для его вычисления. Впрочем, price в данном случае тоже может быть константой, в отличие от VATsumm. Дело в том, что константы определяются в момент написания программы, а при компиляции в код автоматически подставляются их действительные значения. Соответственно, для вычисляемых значений, равно как и для значений, которые может вводить пользователь, нужны не константы, а переменные. Так, если бы константа VAT была переменной, то можно было бы предусмотреть в программе опцию изменения значения НДС.

Переменные определяются при помощи ключевого слова var (от англ. variable):

```
var
  VAT: float;
```

Обратите внимание, что для переменных нужно указывать не только ее имя, но и тип данных. Собственно говоря, константы тоже получают тот или иной тип данных, только это происходит автоматически, в момент компиляции. Так что рассмотри типы данных, предусмотренных в Object Pascal.

ПРИМЕЧАНИЕ

Начиная с Delphi 4, в Object Pascal поддерживаются типизированные константы, значения которых можно изменять по ходу выполнения программы. Объявление констант такого типа производится следующим образом: «const : тип = значение». От обычных переменных они отличаются различиями в обработке компилятором, а так же тем, что для них всегда имеется какое-либо предопределенное значение.

Типы данных

Прежде, чем приступить к рассмотрению типов данных, постараемся понять, для чего это вообще надо. Прежде всего, Object Pascal, равно как и его предшественник – язык Pascal, а так же C и C++ относятся к строго типизированным языкам программирования. Типизированные языки, в отличие от нетипизированных (начиная с BASIC и заканчивая различными языками сценариев вроде JavaScript), имеют то преимущество, что уже на этапе синтаксической проверки кода компилятор не допустит заведомо неверных операций. К примеру, вы не сможете сложить строку с числом. Кроме того, типизация положительно сказывается на быстродействии программ: благодаря тому, что компилятор заранее «знает», что за данные ему следует обработать в каждом конкретном случае, он может подбирать оптимальные инструкции при генерации машинного кода.

Теперь рассмотрим собственно типы данных, имеющиеся в Object Pascal. Прежде всего отметим, что они бывают простыми (числа, символы), структурными (массивы, записи), процедурными и вариантными. Знакомство со структурными и процедурными типами мы отложим до соответствующей главы, а пока что перечислим категории простых типов данных:

- Целочисленные (Integer);
- Вещественные (Real);
- Булевы (Boolean);
- Символьные (Character);
- Строковые (String).

Практически в каждой категории имеется несколько типов данных. В свою очередь это так же вызвано необходимостью более эффективного использования ресурсов компьютера. Например, если для какой-то переменной достаточно диапазона значений от 1 до 100, то зачем для нее выделять память, достаточную для хранения миллиардных значений? Так что начнем подробное с целочисленных типов, как наиболее простых и наглядных.

Всего в современных версиях Delphi предусмотрено 7 различных типов данных для целых чисел, все они приведены в таблице 3.2.

Таблица 3.2. Типы целочисленных данных

Тип	Диапазон	Байт (бит) памяти	Отрицательные значения
Byte	от 0 до 255	1 (8)	Нет
ShortInt	от -128 до 127	1 (8)	Да
Word	от 0 до 65535	2 (16)	Нет
SmallInt	от -32768 до 32767	2 (16)	Да
LongWord	от 0 до 4294967295	4 (32)	Нет
LongInt	от -2147483648 до 21474483647	4 (32)	Да
Int64	от -9223372036854775808 до 9223372036854775807	8 (64)	Да

ПРИМЕЧАНИЕ

> Здесь следует сразу оговориться про понимание памяти в программировании. Так, память считают в байтах. Каждый байт состоит из 8 бит. Бит – это минимальная единица информации, бит может принимать только 2 значения, 0 или 1. Каждая переменная, в зависимости от типа, занимает то или иное количество байт в памяти. Отметим так же, что 2 байта образуют слово (word), а 4 байта – двойное слово.

Помимо перечисленных основных типов, в Delphi имеется еще 2 автоматических целочисленных типа – Integer и Cardinal. Первое, в типичном случае, является синонимом для LingInt, хотя может быть приведено и к типу Int64. Например, если объявит переменную типа Integer и попробовать записать в нее значение, превышающее максимально допустимый размер для типа LongInt, то она автоматически преобразуется в Int64:

```
var x: integer;
...
x: = 21474483647; // здесь x имеет тип LongInt
x: = x + 1; { число 21474483648 не помещается в 4-х байтовый LongInt,
переменная x автоматически приводится к 8-байтовому типу int64 }
```

Что касается Cardinal, то это – устаревшее определение для LongWord, вы сможете встретить его, если будете просматривать исходные коды, написанные во времена первых версий Delphi. Самым распространенным на практике целочисленным типом данных является Integer.

Перейдем к вещественным типам. Для них так же предусмотрены фиксированные типы (правда, в отличие от целочисленных, их не 7, а 6), и один автоматический. Рассмотрим основные типы в таблице 3.3.

Таблица 3.3. Типы вещественных данных

Тип	Диапазон	Байт памяти	Точность
Real48	от ±2.9*10^-39 до 1.7*10^38	6	11-12
Single	от ±1.5*10^-45 до 3.4*10^38	4	7-8
Double	от ±5.0*10^-324 до 1.7*10^308	8	15-16
Extended	от ±3.4*10^-4951 до 1.1*10^4932	10	19-20
Comp	от -2^63+1 до 2^63 -1	8	19-20
Currency	от -922337203685477.5808 до 922337203685477.5807	8	19-20

Имеется так же и автоматический тип – Real, введенный для совместимости с программами, написанными в Delphi 2 или 3. Сейчас тот тип, что был в ранних версиях Delphi, называется Real48 и практически не используется. Вместо него рекомендуется использовать такие типы, как Single или Double. Если же задать тип Real в программе, то он будет автоматически приведен к типу Double.

Нечто подобное можно сказать и про тип Comp – этот 64-разрядный вещественный тип данных изжил себя с момента появления целочисленного типа Int64, и присутствует лишь в целях совместимости со старым программным кодом.

Зато тип Currency активно используется, особенно при работе с базами данных. Как следует из его названия (Currency – валюта), данный тип предпочтительно использовать для финансовых расчетов. И хотя Currency относится к вещественным типам

данных, компилятор обращается с ним как с целым, что позволяет добиться меньших ошибок при округлениях.

В целом, что касается целых и вещественных типов, достаточно запомнить всего две вещи: вещественным типам данных можно присваивать любые численные значения, а целочисленным – только целые:

```
var
x: integer;
y: double;
...
x := 5;
y := 5.25; // обратите внимание, что дробная часть отделяется точкой
y := x + y; // так делать можно
x := x + y; // а так - нельзя, поскольку результатом должно быть целое
```

Булевы, или логические типы данных представлены в Delphi типами Boolean, ByteBool, WordBool и LongBool. Все они отличаются только размером памяти, выделяемым для хранения значения, причем значений может быть только 2 – false (ложь) и true (истина):

```
var x, y: Boolean;
...
x := true;
y := false;
```

Основным типом является 1-байтовый Boolean (он же ByteBool), 2-байтовый WordBool и 4-байтовый LongBool предусмотрены лишь для совместимости в целях взаимодействия с другими языками и платформами.

Что касается символьных типов данных, то в Delphi предусмотрено 2 их типа - ANSIChar и WideChar. Первый является однобайтовым и может хранить в себе 1 символ из множества символов ANSI, коих насчитывается 256. Второй же тип является 2-байтовым и предназначен для хранения 2-байтовых же символов Unicode. Как и в других случаях, Delphi имеет синоним для символьных типов – Char, который на сегодня является аналогом ANSIChar. Что касается присвоения значений, то обычные символы (буквы и цифры) присваивают переменным символьного типа как есть, лишь заключая их в одиночные кавычки. А специальные символы, например, возврат каретки (Enter) назначают при помощи их номера в таблице ANSI, и выделяют знаком решетки:

```
var x, y: Char; // x и y получают тип ANSIChar
...
x := 'a'; // обычные символы
y := #13; // возврат каретки в таблице ANSI имеет номер 13
```

Наконец, еще одним, причем, в общем-то, уже не совсем простым типом данных являются строки. Строковые типы данных отличаются от символьных тем, что могут хранить не единичный символ, а множество символов. В Delphi имеется 3 типа строк: ShortString, AnsiString и WideString. Первый тип строк – ShortString – достался в наследство от языка Pascal и 16-битной Delphi 1.0. Такие строки могут иметь не более 255 символов, и занимают от 2 до 256 байт памяти, в зависимости от размера: Что касается современных строковых типов – AnsiString и WideString, то они могут иметь практически неограниченную длину (AnsiString – до 2 млрд. символов, WideString –

до 1 млрд.) и занимать, соответственно, от 4 байт до 2 Гигабайт памяти. При этом по аналогии с символьными типами, тип AnsiString предназначен для хранения обычных строк, а WideString – для строк в формате Unicode. Ну и еще один тип строк – String является синонимом для типа AnsiString:

```
var str1: ShortString; // короткая строка
var str2: AnsiString; // длинная строка
var str3: String; // тоже длинная строка
...
str1 := 'Начало.'; // Строковые значения заключаются в одинарные кавычки
str2 := 'Конец.';
str3 := str1 + str2; // получим длинную строку, содержащую 'Начало.Конец.'
```

В целом, несмотря на кажущееся разнообразие типов данных, на практике чаще всего ограничиваются всего лишь 5-6 основными типами. Это: Integer, Double, Boolean, Char, String, и иногда – еще и Currency.

Данные и значения

Очевидно, что сами по себе типы данных ничего не означают. Главное их предназначение – хранить те или иные значения, или данные. Так, для хранения числовых данных применяют целочисленные или вещественные типы, в зависимости от того, какого типа числа следует использовать. На практике это означает, что будут применяться типы Integer и Double.

ПРИМЕЧАНИЕ

> Хотя вещественные типы и являются более универсальными, использовать их надо только при реальной необходимости, поскольку они гораздо менее удобны компьютеру для вычислений. Говоря конкретнее, математические операции над вещественными числами выполняются гораздо медленнее, чем над целыми, а ряд иных операций (например, побитовые, инкремент или декремент) вообще недопустимы.

Строковые данные требуют для своего хранения строковых же типов, т.е. String. Исключение может составить тот случай, когда следует сохранить один и только один символ – в таком случае предпочтительно (а иногда – необходимо) использовать тип Char. Ну а в том случае, если требуется привести какое-то значение к однозначному выводу, используют логический тип Boolean. Например, если сравнивать одно число с другим на предмет того, является ли оно больше, то результатом сравнения будет либо «да», либо «нет», т.е. true или false:

```
var
  x: boolean;
...
x := 5 > 6; // получаем false, т.к. 5 не больше 6
```

Важно отметить, что переменные одного типа могут хранить данные только такого же типа. Кроме того, произведение каких-либо манипуляций над данными возможно только в том случае, если они относятся к одной категории. Исключение составляют разве что такие пары, как строки–символы и вещественные–целые числа. При этом результирующая переменная должна иметь тип строки для первой пары и вещественного числа – для второй. Впрочем, подробнее все это будет рассмотрено в контексте операций языка Object Pascal.

Операции и их типы

В любом языке программирования имеются знаки операций. Кроме того, некоторые ключевые слова, например такие, как div или mod также обозначают операции. Все операции в Object Pascal можно разделить на следующие типы: логические, арифметические, логические, операции присвоения и отношения, а так же специальные операции. Для их обозначения используются математические символы или ключевые слова. Участвующие в операциях значения (переменные) называются операндами. При этом та или иная операция может работать с операндами определенного типа. А результатом может быть данные как такого же типа, та и другого (например, для того же сравнения).

Начнем ознакомление с арифметических операций, как наиболее распространенных (не только и не столько в программировании, сколько в реальной жизни). С полным их перечнем, а так же с типами исходных и результирующих данных можно ознакомиться в таблице 3.4.

Таблица 3.4. Арифметические операции

Операция	Название, тип	Описание	Операнды	Результат
+	Сложение, бинарная	Возвращает сумму левого и правого операндов	integer, real	integer, real
-	Вычитание, бинарная	Возвращает разницу левого и правого операндов	integer, real	integer, real
*	Умножение, бинарная	Возвращает произведение левого операнда на правый операнд	integer, real	integer, real
/	Деление, бинарная	Возвращает результат деления левого операнда на правый операнд. Результат может быть дробным	integer, real	real
mod	Остаток от деления, бинарная	Возвращает остаток от деления левого операнда на правый операнд	integer	integer
div	Деление нацело, бинарная	Возвращает целую часть числа, получившуюся в результате деления	integer	integer
-	Унарный минус	Возвращает число, противоположное операнду	integer, real	integer, real
+	Унарный плюс	Явно указывает знак числа	integer, real	integer, real

Можно отметить, что практически для всех арифметических операций результатом будут данные того же типа, что и операнды. Единственное исключение – это операция деления, результатом которой всегда будет вещественное число.

Другой распространенный тип операций – логические. В Object Pascal имеются все 4 типа логических операций: не, и, или, исключающее или (таблица 3.5).

Таблица 3.5. Логические операции

Операция	Название	Описание
not	Логическое отрицание (НЕ)	Возвращает false, если выражение может быть приведено к истине, в противном случае возвращает true

and	Логическое (И)	Возвращает true, когда оба выражения истинны. В противном случае возвращает false
or	Логическое (ИЛИ)	Возвращает true, когда хотя бы одно из выражений истинно. В противном случае возвращает false
xor	Логическое (исключающее ИЛИ)	Возвращает true, когда только одно из выражений истинно. В противном случае возвращает false

Варианты возвращаемых значений для логических операций приводятся ниже:

```
not true          // возвращает false
not false         // возвращает true
true and true     // возвращает true
true and false    // возвращает false
false and false   // возвращает false
true or true      // возвращает true
true or false     // возвращает true
false or false    // возвращает false
true xor true     // возвращает false
true xor false    // возвращает true
false xor false   // возвращает false
```

Логическому сравнению подлежат не только булевские значения, но и любые другие выражения, которые могут быть к ним приведены. Например, выражение «3=4» может быть использовано в качестве логически сравниваемой единицы, поскольку результатом его оценки будет булево значение ложь (false).

Те же самые знаки операций, что используются в логических операциях, задействованы и в другом типе операций – побитовых. Побитовые операции выполняются над числами, представленными в двоичном виде (т.е. только нули и единицы). Однако сами операнды могут быть десятичными, шестнадцатеричными, или восьмеричными целыми числами. Например, десятичное число 5 представляется как двоичное 101, десятичное 6 – как 110, а шестнадцатеричное F3 – как двоичное 11110011.

Хотя побитовые операции выполняются над двоичными данными, возвращаемые значения являются стандартными числами. Список всех побитовых операций приводится в таблице 3.6.

Таблица 3.6. Побитовые операции

Операция	Название	Описание
and	Побитовое И	Возвращает число, являющееся результатом побитового сравнения «И»
or	Побитовое ИЛИ	Возвращает число, являющееся результатом побитового сравнения «включающее ИЛИ»
xor	Побитовое исключающее ИЛИ	Возвращает число, являющееся результатом побитового сравнения «исключающее ИЛИ»
not	Побитовое НЕ	Возвращает число, с битами, расположенными в обратном порядке
shl	Сдвиг влево	Сдвигает первый операнд влево на число разрядов, заданных вторым операндом. Освобождающиеся правые биты заполняются нулями
shr	Сдвиг вправо	Сдвигает первый операнд вправо на число разрядов, за-

данных вторым операндом. Освобождающиеся левые биты отбрасываются

Чтобы явно представить себе, как работают побитовые операции, обратимся к следующему примеру. Допустим, имеется 2 переменных – x и y:

```
var x, y: integer;
...
x := 3;
y := 5;
```

В двоичном представлении число 3 будет выглядеть как 0011, а 5 – как 0101. Теперь посмотрим, какие результаты даст каждая из побитовых операций сравнения и операции отрицания над этими числами:

```
x or y      // Получим 7:   0011 | 0101 = 0111
x and y     // Получим 1:   0011 & 0101 = 0001
x xor y     // Получим 6:   0011 ^ 0101 = 0110
not x       // Получим 12:  ~0011 = 1100
```

Что касается операций побитового сдвига, то они дадут следующие результаты:

```
y shl 2     // Получим 20:  0101 shl 2 = 10100
y shr 2     // Получим 1:   0101 shr 2 = 01
-y shr 2    // Получим -2:  -0101 shr 2 = -10
```

Теперь нам остается исследовать еще один тип операций – операции сравнения. Эти операции всегда требуют наличия двух операндов: они сравнивают значения левого и правого операндов, и возвращают результат сравнения в виде логического значения, которое может принимать значение false или true (ложь или истина). Все имеющиеся в Object Pascal операции сравнения приведены в таблице 3.7.

Таблица 3.7. Операции сравнения

Операция	Название	Описание	Пример, дающий true
=	Равно	Возвращает истину (true), когда левый и правый операнды равны.	1==1
<>	Не равно	Возвращает истину, когда левый и правый операнды не равны.	1!=2
>	Больше	Возвращает истину, когда левый операнд больше правого.	2>1
<	Меньше	Возвращает истину, когда левый операнд меньше правого.	1<2
>=	Больше или равно	Возвращает истину, когда левый операнд больше правого или равен ему.	1>=0 1>=1
<=	Меньше или равно	Возвращает истину, когда левый операнд меньше правого или равен ему.	0<=0 0<=1

Операции сравнения могут работать с различными типами данных, включая строки, при этом для строк используется стандартный подход: символы последовательно сравниваются по своим ASCII-кодам, или Unicode-кодам, если используется строковая переменная в формате WideChar.

К операциям сравнения так же можно отнести операцию членства – in. При этом в качестве левого операнда может выступать данные ординарного типа (Byte или

Char), а в качестве левого – подмножество значений, совместимых с данным типом. Например, если имеется переменная x типа Char, то можно проверить, является ли ее значение частью некоего набора символов:

```
var
  x: Char;
  z: Boolean;
...
x := 'b';
z := x in ['a'..'d'];
```

В данном случае в качестве результата (z) мы получим истину, поскольку символ b является членом указанного множества ['a'..'d'], в которое входят символы a, b, c и d.

Наконец, в Object Pascal имеется еще 2 операции – as и is. Они служат для приведения типа и проверки типа, соответственно. Например, если мы хотим проверить, является ли некая переменная «x» целым, то можно написать такое выражение:

```
b := x is Integer; // b получит значение true, если x - целое
```

Ну а операция as используется для приведения данных одного типа к другому, причем, преимущественно, при работе с объектами:

```
ObjA as ObjB;
```

При использовании операции as следует быть осторожным, поскольку приведение одного типа к другому возможно далеко не всегда.

Выражения и приоритет операций

Как нам уже известно, выражения – это, прежде всего, набор данных, переменных и операторов. Развивая это определение, можно сказать, что выражения состоят из значений и операций над ними, например:

```
a := b + c;
d := e * f;
g := a - d div 2;
```

Так же, как и в обычной математике, при составлении выражений в Object Pascal, следует учитывать приоритет выполнения операций. Например, операции умножения или деления должны выполняться раньше, чем сложение и вычитание. Так, в 3-й строке из приведенных выше примеров выражений, согласно математическим правилам, сначала выполняется операция деления нацело (d div 2), затем результат этой операции вычитается из a, и итоговое значение присваивается переменной g. Все те же правила действуют и в программировании, но поскольку перечень операций не ограничивается арифметическими, а в рамках одного выражения могут быть использованы различные типы операций, то было бы неплохо внести полную ясность в этот вопрос.

Прежде всего, приоритет выполнения определяется принадлежностью конкретной операции к тому или иному типу. Так, первыми выполняются унарные операции, затем – операции умножения (включая деление и побитовые), далее обрабатываются сложение и вычитание (опять, таки включая побитовые or и xor), и, наконец, в последнюю очередь – операции отношения. Для удобства представим все операции, классифицированные по уровню приоритета, в виде таблицы (табл. 3.8).

Таблица 3.8. Приоритет выполнения операций

Операторы	Уровень приоритета	Категория
@, not	Высший	Унарные
*, /, div, mod, and, shl, shr, as	Высокий	Умножение
+, -, or, xor	Средний	Сложение
=, <>, >, <, <=, >=, in, is	Низкий	Отношение

В тех случаях, когда порядок следования операций, задаваемый их приоритетом, следует изменить, применяют круглые скобки. Например, если мы имеем выражение:

```
g := a - d div 2;
```

И нам требуется сначала выполнить вычитание, то достаточно заключить участвующие в операции вычитания операнды в круглые скобки:

```
g := (a - d) div 2;
```

Таким образом, сначала здесь из a вычитается d, и лишь затем производится операция деления нацело.

Структура программы

Теперь, когда мы уже знаем об основных языковых конструкциях языка – «словах» (ключевые слова, переменные, операторы и т.д.) и выражениях, рассмотрим, как все это объединяется в общий код программы.

Итак, программа состоит из: заголовка, за которым следуют список подключаемых модулей, объявления меток, констант, типов данных и переменных, описания процедур и функций. Вслед за этой заголовочной частью располагаются собственно программные инструкции. На практике все это выглядит таким образом:

```
program <Имя программы>;
uses <Список модулей>;
label <Список меток>;
const <Список констант>;
type <Описания типов>;
var <Объявления переменных>;
<Описание процедур и функций>;
begin
<Инструкции>;
end.
```

В структуре той или иной программы часть этих разделов может быть опущена за ненадобностью. Обязательными являются всего 3 ключевых слова – program (вместе с названием), а так же begin и end. Разумеется, между begin и end должны находиться какие-либо инструкции. Так, возвращаясь к примеру «Hello, World», в нем можно обнаружить лишь название программы и блок инструкций:

```
program hello; // название
{$APPTYPE CONSOLE} //это указание компилятору к коду программы не относится
begin  // начало исполняемого кода
  write('Hello, World!'); // инструкции
  readln;
end.  // конец программы
```

ПРИМЕЧАНИЕ

> Список модулей в данном случае не нужен, поскольку мы не используем никаких дополнительных функций или процедур из библиотеки Object Pascal. Но, разумеется, в более сложных программах они могут понадобиться.

Так же следует отметить, что те или иные модули могут повторяться, причем неоднократно. Например, объявления констант или переменных могут чередоваться, равно как и описание типов, функций, или процедур. Другое дело, что с точки зрения собственного удобства следует следить за тем, чтобы все это было педантично разложено по своим местам, поскольку в противном случае много времени будет уходить на поиск того или иного объявления.

Рассмотренная нами структура характерна для программ на Pascal в целом, в том числе и для Windows-приложений, разрабатываемых визуальными средствами. Структура же отдельных модулей, представляющих собой описание того или иного окна, имеет ряд отличий. В общих чертах ее можно представить следующим образом: начинается такой модуль с ключевого слова unit, после которого следуют секции interface и implementation. Иногда могут быть также использованы еще 2 секции - initialization и finalization. Ну а завершается все это, разумеется, ключевым словом end с точкой:

```
unit <Имя модуля>;
interface
<описания и объявления переменных, констант, типов, классов и т.д.>
implementation
<Инструкции>;
initialization
<Опционально, инструкции для выполнения при загрузке программы>;
finalization
<Опционально, инструкции для выполнения при завершении программы>;
end.
```

При создании новых форм в среде Delphi весь необходимый код создается автоматически, разве что опциональные (и довольно редко используемые на практике) секции загрузки или завершения, при необходимости, придется дописывать самостоятельно.

Завершая эту тему, затронем еще один аспект структуры программ – подпрограммы. В качестве подпрограммы понимается блок инструкций, предназначенный для решения какой-либо частной задачи в рамках самой программы. Такие блоки в Object Pascal могут просто выполняться – в таком случае они называются процедурами, либо выполняться и возвращать какой-либо результат – в таком случае они называются функциями. И в том, и в другом случае, они могут принимать какие-либо значения для обработки. Эти значения называются аргументами. В целом же синтаксис обращения к процедуре или функции выглядит следующим образом:

```
<Имя процедуры или функции>[(список аргументов)];
```

Подробнее об этом будет рассказано в соответствующей главе, пока же отметим для себя сам факт их существования, поскольку очень скоро нам придется столкнуться с рядом процедур и функций, реализованных в самом языке Object Pascal.

Операторы Object Pascal

Знание синтаксиса языка лишь теоретически позволяет приступить к созданию программ. Дело в том, что важнейшей частью любого языка программирования, во многом определяющими удобство составления алгоритмов, являются его управляющие структуры – операторы, или инструкции.

Об управляющих структурах

В реальных программах выполнение операций не бывает строго последовательным: постоянно требуются различные переходы, ветвления, повторения и т.д. Так, если мы обратимся к блок-схеме даже такого простого алгоритма, который требуется для игры «угадай число» (см. рис. 1.1), то увидим там целых два ветвления, причем оба с переходом.

Собственно за переход, в классическом варианте, отвечает небезызвестная инструкция безусловного перехода goto, которая в Pascal используется совместно с метками, декларируемыми в заголовочной части программы при помощи ключевого слова label. Использование инструкции безусловного перехода восходит корнями к тем временам, когда создавались первые высокоуровневые языки программирования, в том числе и Pascal. Она досталась им в наследство от низкоуровневых языков типа Assembler, в которых описание программы создавалось в виде, удобным для машины. Но на сегодня такой подход уже не востребован и вышел из употребления, вместе с безусловным переходом и инструкцией goto.

СОВЕТ

> За 10 лет, прошедших с того момента, как я последний раз использовал язык Basic, мне ни разу не довелось ни использовать инструкцию goto в собственных программах, ни встретить ее в миллионах строк просмотренного кода. Вывод: она просто не нужна в Object Pascal, так что старайтесь не использовать goto!

Инструкция goto относится к т.н. «простым операторам». К ним же относится пустой оператор, который вообще ничего не делает. Он может находиться в любом месте программы, где допускается, или требуется наличие какой-либо инструкции. Именно по второй причине, т.е. при необходимости использования инструкции в том или ином месте программы (по правилам языка), и скрывается вся ее ценность: поскольку пустая инструкция ничего не делает, то она является идеальной «заглушкой» в подобных ситуациях. Обозначается пустой оператор знаком «точка с запятой».

В тех же случаях, когда наоборот, по правилам языка можно использовать всего лишь одно выражение или оператор, а нужно несколько, используют составные операторы. Составной оператор представляет собой группу из произвольного числа любых инструкций, ограниченных ключевыми словами begin и end:

```
begin
  <инструкция 1>;
  <инструкция 2>;
  ...
  <инструкция N>;
end;
```

Сколько бы ни входило инструкций в такой блок, для компилятора он воспринимается как единое целое и может располагаться в любом месте программы, где допускается наличие хотя бы одного оператора.

Составные операторы могут вкладываться один в другой, при этом глубина таких вложений в Object Pascal не ограничена.

Условный оператор if

Пожалуй, самой важной инструкцией для управления выполнением программы является условный оператор if. Именно он отвечает за ветвление, т.е. выполнение (или невыполнение) того или иного варианта кода в зависимости от условий. Оператор if используется совместно с ключевым словом then, а в том случае, когда предусмотрен альтернативный вариант выполнения – еще и с else. В целом синтаксис инструкции получается следующим:

```
if <условие> then <код> [else <альтернативный код>]
```

В качестве условия может быть использовано любое выражение, которое может быть приведено к булевскому значению, т.е. к false или true. Как правило, это бывают операции сравнения, например:

```
if a > 5 then b := 10;
if x <> 0 then y := 1 else y :=2;
```

В первом случае, если переменная a больше 5, то переменной b будет присвоено значение 10, если же a меньше или равно 5, то ничего выполнено не будет и управление будет передано следующему выражению. Во второй строке переменная x проверяется на ненулевое значение, и если x окажется числом, отличным от 0, то переменной y будет присвоено значение 1, в противном случае (если x равно 0) переменной y будет присвоено значение 2.

В тех случаях, когда для того или иного варианта кода требуется использовать более одного выражения, используют составной оператор:

```
if a > 5 then
  begin
    b := 10;
    c := 20;
  end;
```

Как мы уже знаем, в соответствии с правилами синтаксиса Object Pascal, все то, что помещено между ключевыми словами begin и end, равно как и сами эти слова, воспринимаются как 1 оператор. Обратите внимание, что в конце поставлен пустой оператор – точка с запятой. В данном случае по правилам синтаксиса он здесь не обязателен, однако будет хорошим тоном завершать им каждый составной оператор, чтобы выделить тем самым окончание той инструкции, в которой он был применен. В данном случае получается, что мы использовали 2 оператора – составной и пустой, однако нарушения синтаксиса тут нет – компилятор сочтет, что пустой оператор следует уже после условного. Но если бы мы использовали еще и блок else, это привело бы к ошибке синтаксиса, поскольку между then и else допустима лишь 1 инструкция. Поэтому в таком случае точку с запятой следует разместить уже после оператора, следующего за else:

```
if a > 5 then
```

```
  begin
    b := 10;
    c := 20;
  end
else
  begin
    b := 20;
    c := 15;
  end;
```

В тех случаях, когда требуется предусмотреть 3 или более вариантов исполнения, используют вложение операторов if друг в друга. Например, если требуется выполнить один вариант когда, когда некая переменная x меньше нуля, другой – если x равна 0, и третий – если x больше нуля, то синтаксис операторов может быть следующим:

```
if x < 0 then <вариант для x<0> else
  if x = 0 then <вариант для x=0> else <вариант для x>0>;
```

В данном случае использован вложенный оператор if, который выполняется в случае, когда переменная x не меньше 0. Он проверяет, не является ли значением x число 0, и если нет, то, учитывая, что x явно не меньше, чем 0 (это условие к моменту выполнения вложенного оператора if уже проверено внешним, т.е. первым в данном выражении оператором if), значит значение x больше 0.

Оператор выбора case

Условный оператор удобен в тех случаях, когда необходимо проверить 1-2-3 варианта. При большем числе получается слишком громоздкая и неудобная для восприятия конструкция из множества вложенных инструкций. Скажем, если требуется проверить 5 значений переменной x, то получим такую конструкцию:

```
if x = 1 then ;
  else if x = 2 then ;
    else if x = 3 then ;
      else if x = 4 then ;
        else if x = 5 then ;
```

Очевидно, что код получается слишком громоздким, и малоэффективным. В таких случаях на помощь приходит семафор – оператор множественного выбора case. Он состоит из выражения, являющегося селектором, списка вариантов, представленного константами или значениями, и необязательной части else. Таким образом, формат оператора case таков:

```
case <выражение-селектор> of
  <значение 1>: <код для значения 1>;
  <значение 2>: <код для значения 2>;
  ...
  <значение N>: <код для значения N>;
  [else <код для непредусмотренных явно значений>;]
end
```

Единственным ограничением семафора, в сравнении с условным оператором, является то, что в качестве селектора могут выступать лишь данные порядкового типа, ска-

жем, целым числом или же символом. Впрочем, для подавляющего числа случаев этого достаточно. Например, приведенный выше вариант кода с 4 вложенными условными операторами, при помощи case можно оформить так:

```
case x of
  1: ;
  2: ;
  3: ;
  4: ;
  5: ;
end;
```

Здесь подразумевается, что типом переменной x является целое число, поскольку тип значений, коими в данном случае являются целые числа, должен соответствовать типу селектора.

Инструкция выбора выполняется следующим образом: вначале, при необходимости, вычисляется значение селектора, затем производится последовательный обход вариантов на предмет совпадения с селектором. В случае совпадения, выполняется инструкция, предусмотренная для этого варианта, после чего выполнение оператора выбора заканчивается. Если же ни один из перечисленных вариантов не совпал со значением селектора (для нашего случая – если x меньше 1 или больше 5), то либо оператор завершается без каких-либо действий, либо, при наличии блока else, выполняются заданные в нем инструкции.

В качестве констант выбора могут выступать не только единичные значения, но и их список, разделенный запятыми, или же диапазоны, определенные границами из 2 констант, разделенных двумя точками. В таком случае мы можем объединить логически связанные значения в группы, для которых следует выполнить один и тот же код. Например, таким образом можно получить название времени года в зависимости от порядкового номера месяца (листинг 4.1).

Листинг 4.1. Использование оператора case

```
var
  month: integer;
  season: string;
...
case month of
  1,2,12: season := "зима";
  3..5:   season := "весна";
  6..8:   season := "лето";
  9..11:  season := "осень";
  else    season := "других не знаем!";
end;
```

В данном случае, если переменная month имеет значения 1, 2 или 12, то переменной season присваивается значение «зима», если же значение переменной month окажется в диапазоне от 3 до 5 (включительно), то season получит значение «весна», и т.д.

Оператор цикла for

Для написания практически любой программы, помимо операторов условия, требуются операторы цикла, и в Object Pascal, они, разумеется, есть. Прежде всего, это оператор цикла с параметром – for. Такой тип цикла обычно применяют в тех случаях, когда количество возможных повторов известно заранее. Он имеет 2 варианта написания: один – для цикла с приращением, и другой – для цикла с уменьшением:

```
for <параметр> := <выражение 1> to <выражение 2> do <тело цикла>;
for <параметр> := <выражение 1> downto <выражение 2> do <тело цикла>;
```

В первом случае (с использованием цикла for-to) при каждом проходе цикла, называемом итерацией, значение параметра увеличивается на 1, а во втором (for-downto) – уменьшается на 1. При этом в качестве начального значения используется «выражение 1», а в качестве конечного – «выражение 2». Разумеется, если для цикла to значение первого выражения изначально будет больше значения второго, или наоборот, меньше (для цикла downto), то цикл не будет выполнен ни разу.

Практическое применение циклов крайне разнообразно. Если привести наиболее общий пример из программирования, то цикл – идеальный способ заполнения массива. Например, если требуется организовать цикл для заполнения массива из 10 числовых значений последовательно возрастающими числами, то можно записать:

```
for i := 0 to 9 do MyArray[i]=i;
```

В данном случае элементам массива MyArray последовательно назначаются значения от 0 до 9.

ПРИМЕЧАНИЕ

> Сами массивы будут рассмотрены несколько позже, в главе, посвященной структурным типам данных.

Теперь рассмотрим цикл for с отрицательным приращением на примере вычисления математического факториала (последовательное произведение всех целых чисел от 1 до самого числа). Для этого нам понадобится следующий цикл:

```
var num, rez: integer;
...
rez := 1;
for num := num downto 1 do rez := rez * num;
```

Здесь нам потребовалась небольшая подготовительная работа: определена переменная rez, в которой будет храниться вычисляемое значение, и ей присвоено значение 1. В качестве числа, для которого вычисляется факториал, выступает переменная num, она же используется для самого цикла в качестве счетчика. Поскольку нам надо будет прекратить выполнение цикла, после того, как счетчик (num) достигнет значения 1, то именно это значение и указано в качестве конечного условия.

В итоге, если переменной num присвоить значение 5, то после прохождения цикла переменная rez получит значение 120. Хотя в результате работы такого цикла получится выполнение как бы наоборот (т.е. не вместо 1*2*3*4*5, на самом деле выполняется 5*4*3*2*1), это никак не помешает получить верный результат.

Наконец, в качестве тела цикла, как и в случае с уже рассмотренными операторами, может использоваться составной оператор. Кроме того, циклы могут быть вложены один в другой, при этом важно лишь следить за тем, где заканчивается вложенный цикл и начинается внешний. Для этого полезно следить за правильным оформлением программы, в частности, использовать отступы, в качестве которых можно использовать либо знак табуляции, либо пробелы, в последнем случае их желательно ставить не менее 2 (листинг 4.2).

Листинг 4.2. Вложенные циклы и форматирование кода

```
for x := 5 to 10 do begin
  z := x;
  for y := 10 to 20 do begin
    z := z + x * y;
    writeln(z);
  end; // конец вложенного цикла
  writeln(x);
end; // конец внешнего цикла
```

При использовании циклов с параметром важно помнить, что изменение значения параметра в теле цикла недопустимо.

Операторы циклов while и repeat

Помимо классического цикла с параметром, в Object Pascal предусмотрено еще 2 вида циклов – с предусловием и с постусловием. В качестве цикла с предусловием выступает оператор while. Он имеет следующий синтаксис:

```
while <Условие> do <тело цикла>
```

Этот цикл будет выполняться до тех пор, пока верно выражение, указанное в качестве условия. Соответственно, если значение выражение будет изначально ложным, то цикл не будет выполнен ни разу, например:

```
while false do ;
```

В то же время, при помощи оператора while удобно делать бесконечный цикл. В бесконечных циклах весь контроль возлагается на операторы, помещаемые в тело цикла:

```
while true do begin
  <инструкции>
end;
```

ПРИМЕЧАНИЕ

> Очевидно, что цикл с условием, которое изначально истинно и никак не изменяется, будет выполняться вечно. Таким образом, в теле цикла следует предусмотреть возможность его прерывания иным способом.

В отличие от while, цикл с постусловием, задаваемый при помощи оператора repeat, всегда выполняется хотя бы 1 раз, поскольку проверку на соответствие условию он проходит после того, как будет выполнено его тело:

```
repeat <тело цикла> until <условие>
```

Важно так же отметить, что в цикле с постусловием ключевые слова repeat и until образуют как бы составной оператор. Иначе говоря, если в цикле while при необхо-

димости использовать более одной инструкции следует использовать составной оператор, то для цикла repeat этого не требуется:

```
repeat
  x := x + 1;
  y := x * 2;
until y < 1000;
```

При использовании циклов while и repeat важно помнить, что хотя бы одна из инструкций, выполняемых в теле цикла, должна влиять на значение его условия, в противном случае вы рискуете получить зацикливание.

Операторы break и continue

Мы уже поднимали вопрос бесконечного цикла и возможного зацикливания программы, чреватого ее зависанием. Чтобы иметь возможность обработки подобных ситуаций, а так же сделать сами циклы более гибкими, используют специальные операторы – break и continue. Оператор break позволяет завершить цикл досрочно, а оператор continue – выполнить только часть операторов в теле цикла, перейдя к его следующей итерации.

Так, досрочный выход из цикла, определяемый при помощи оператора break, происходит в месте вызова этого оператора, и управление будет передано первому оператору, находящемуся после цикла. Например, в уже рассмотренном нами бесконечном цикле, можно задать условие его прерывания таким образом:

```
while true do begin
  a := b * c;
  if a > 1000 then break;
  b := b + 1; // в случае если a > 1000, эта строка выполнена не будет
end;
```

Еще одним полезным применением досрочного выхода является его использования в качестве дополнительного параметра. Например, если нам нужен цикл, который должен прерваться по 1 из 2 условий, то для второй проверки мы можем использовать условный оператор if совместно с break:

```
repeat
  i := i + 1;
  if i > 100 then break;
  y := y - 1;
until y < 50;
```

Здесь мы определили цикл, который будет завершен либо после того, как значение переменной y достигнет 50 (что задано в самом условии цикла), либо если значение переменно x превысит указанное в условии if значение 100 – здесь как раз будет задействован оператор break.

Иногда бывает необходимо перейти к следующему шагу цикла досрочно, пропустив часть операторов. Для этих целей используют оператор continue. В отличие от break, этот оператор не завершает цикл, а заставляет программу досрочно перейти к новой проверке условия цикла. Рассмотрим эту ситуацию на примере. Допустим, что нам надо получить список чисел, на которые число 100 делится без остатка:

```
for i := 1 to 100 do begin
```

```
  if 100 mod i <> 0 then continue;
  writeln(i);
end;
```

Для этого мы определили цикл, в котором при каждой итерации делитель, в качестве которого выступает счетчик цикла, увеличивается на единицу. В самом теле цикла использован условный оператор, в котором условием выступает выражение, в котором производится операция «остаток от деления» и ее результат сравнивается с 0. Таким образом, если дано условие истинно (т.е. если остаток равен нулю), то интерпретатор переходит к следующему оператору в теле цикла, который выводит нужное нам число, в противном случае выполняется оператор continue и начинается новый шаг цикла.

Операторы в действии

Таким образом, мы рассмотрели практически все операторы языка Object Pascal, включая 3 типа стандартных циклов. Еще один цикл – for-in, появившийся в Delphi 2005, является частным случаем цикла for-to и позволяет без лишних усилий создавать циклы для данных порядкового типа. Тем не менее, останавливаться на возможностях языка, появившихся после Delphi 7, мы здесь не будем. Рассмотрим лучше практическое применение изученных операторов для реализации алгоритмов, для чего вновь обратимся к игре «угадай число».

Игра выполняется до тех пор, пока пользователь не угадает число из заданного диапазона (скажем, от 0 до 100). При этом, если предложенный пользователем ответ больше или меньше загаданного программой числа, то программа должна выводить соответствующую подсказку. Соответственно, для реализации этой программы нам понадобятся следующие операторы:

- Цикл while, выполняющийся до тех пор, пока число не угадано
- Два условных оператора if, которые будет проверять, не является ли ответ большим или меньшим, и выводить соответствующую подсказку

Помимо них, нам потребуются несколько стандартных функций Object Pascal – для вывода информационных сообщений, для ввода пользователем значений, а так же для «загадывания» числа. Ввод и вывод в консольных программах осуществляется при помощи функций read и write, соответственно. А для генерации псевдослучайных чисел используют процедуру randomize и функцию random.

ПРИМЕЧАНИЕ

> Несколько позже мы подробно ознакомимся с тем, что такое процедуры и функции, и чем они различаются. Пока же достаточно просто принять к сведению, что перечисленные здесь функции будут делать в данном случае.

Итак, для начала запустите Delphi и создайте новое приложение командной строки (File → New → Other → New/Console Application). После того, как Delphi создаст новый проект, сразу сохраните его под каким-либо осмысленным именем, попутно выбрав для него подходящее место на диске. Назовем эту программу «Ugadaika» и помести ее в одноименный каталог на диске C:.

Теперь можно приступать к написанию кода. Поскольку программа должна начинаться с деклараций переменных, то с них и начнем. Нам понадобятся 2 переменных: одна – для хранения загаданного числа, и вторая – для получения ответа пользователя. Назовем их a и b и поместим объявление перед ключевым словом begin. Таким образом, в самом начале работы программа примет такой вид, как показано на примере листинга 4.3.

Листинг 4.3. Программа «угадай-ка» в самом начале разработки

```
program ugadaika;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  a,b: integer;
begin

end.
```

Теперь можно приступать к написанию основного кода, который, как мы знаем, должен располагаться между begin и end. Начинаться он должен с загадывания числа. Для этого поместим в начало программы (начиная со строки, следующей за begin), следующий код:

```
randomize; // инициализация генератора псевдослучайных чисел
a := random(100)+1; // присваивание переменной a значения от 1 до 100
```

Далее следует подготовиться к циклу отгадывания, для чего надо, во-первых, установить переменную b в заведомо неподходящее значение (скажем, в 0), а во-вторых – вывести пользователю сообщение, в котором пояснить, чего ему надо делать:

```
b:=0;
write('Input a number between 1 to 100 and hit Enter');
```

Здесь мы использовали английские символы по той причине, что кодовые номера символов кириллицы для DOS и Windows различаются. Соответственно, чтобы опустить здесь процедуры перекодировки символов, мы пошли по пути наименьшего сопротивления и ограничились латиницей.

Следующим этапом будет создание самого цикла. Мы уже определили условие его выполнения – до тех пор, пока число не отгадано, т.е. пока a не равно b. Соответственно, выглядеть он будет так:

```
while (a<>b) do
```

Кроме того, поскольку в цикле будет явно больше 1 оператора, сразу определим составной оператор, добавив begin и end. В результате общий вид программы на текущем этапе получится таким, как показано на листинге 4.4.

Листинг 4.4. «Угадай-ка» с основными блоками

```
program ugadaika;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
```

```
  a,b: integer;
begin
  randomize;
  a := random(100)+1;
  b := 0;
  write('Input a number between 1 to 100 and hit Enter');
  while (a<>b) do begin

  end;
end.
```

Теперь займемся телом цикла. Для начала нам надо предоставить пользователю предложить свой вариант ответа. Однако не будем забывать, что пользователь – тоже человек, и не помешало бы вывести подсказку перед точкой ввода числа, хотя бы самую простую, состоящую из знака вопроса с двоеточием:

```
write(#13+#10+'?:');
```

Использованные здесь символы #13 (возврат каретки) и #10 (новая строка) переносят точку ввода на одну строку вниз для лучшего эстетического восприятия. Теперь дадим пользователю возможность ввести свой вариант ответа, воспользовавшись функцией read:

```
read(b); // введенное число будет помещено в переменную b
```

Теперь остается написать условные операторы, проверяющие введенное число на соответствие загаданному и выдающими нужную подсказку. Начнем со случая, когда введенное пользователем число больше загаданного:

```
if (b>a) then write('Too much!');
```

Теперь составим условие для противоположного случая:

```
if (b<a) then write('Too small!');
```

Вот, в общем-то, и все! Остается после завершения цикла предусмотреть вывод сообщения о том, что число отгадано, для чего вновь воспользуемся функцией write, а заодно – функцией sleep, чтобы предотвратить преждевременное закрытие окна:

```
write('Cool! You win!');
sleep(2000); // пауза в 2 секунды
```

В итоге мы получим вполне работоспособную программу, полный исходный код которой приведен в листинге 4.5, а на CD он находится в Demo\Part1\Ugadaika.

Листинг 4.5. Полный исходный код игры

```
program ugadaika;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  a,b: integer;
begin
  randomize;
  a := random(100)+1;
  b := 0;
```

```
  write('Input a number between 1 to 100 and hit Enter');
  while (a<>b) do begin
    write(#13+#10+'?:');
    read(b);
    if (b>a) then write('Too much!');
    if (b<a) then write('Too small!');
  end;
  write('Cool! You win!');
  sleep(2000);
end.
```

Структурные типы данных

Теперь, когда мы уже знаем основы языка и даже можем писать небольшие программы, можно перейти к детальному изучению предоставляемых Delphi возможностей для работы с самыми различными типами данных – массивами, записями, множествами и т.д. Здесь же мы рассмотрим вопросы преобразования типов данных.

Пользовательские типы данных

При разработке программ довольно часто оказывается недостаточно тех типов данных, которые представлены языком программирования. Например, бывает удобным совместить в одной переменной сразу ряд однотипных данных, или же предусмотреть хранение данных разных типов, например, строк и чисел. К счастью, в Object Pascal имеется возможность создавать собственные типы данных на основе уже имеющихся, совмещая их, или комбинируя. Например, для создания упорядоченного списка однотипных данных используют массивы (arrays), а для объединения нескольких типов в один – записи (records).

Другим аспектом применения собственных типов данных, упрощающих процесс программирования и делающий его более понятным человеку, является использование множеств – именованного набора из нескольких возможных значений.

Создание того или иного типа данных всегда начинается с декларации, или описания нового типа данных. Делается это в заголовочной части программы или модуля и начинается с ключевого слова type. После того, как новый тип данных определен, можно создавать переменные нового типа – точно так же, как и любого простого. Ну а особенности работы с тем или иным пользовательским типом данных определяются тем, к какому виду он принадлежит (массив, запись и т.д.), и собственно его реализацией (размер и размерность для – массивов, набор других типов – для записей и т.п.).

Массивы

Массив – это упорядоченная структура, состоящая из множества однотипных элементов, имеющих общее имя. Таким образом, при помощи всего лишь одной переменной можно хранить целый набор данных, при этом каждый элемент массива имеет свой собственный индекс, или индексы – в случае, если массив многомерный. Массив объявляется в Object Pascal при помощи ключевого слова Array, необязательных квадратных скобок с указанием числа элементов массива, и типа данных для элементов массива:

```
Array [индексы] of <тип элементов>;
```

Количество индексов определяет размерность массива. Так, для одномерного массива (в математике – вектор) требуется всего один индекс, а для двумерного массива (матрицы) понадобится 2 индекса и т.д. Объявления массивов могут выглядеть так:

```
type MyArray1 = array [1..100] of integer;
type MyArray2 = array [1..10, 1..10] of integer;
```

В первом случае определен одномерный массив на 100 элементов типа целых чисел, во втором – двумерный массив размерностью 10 на 10, т.е. так же на 100 элементов типа целых чисел. После того, как массив определен, можно создавать переменные соответствующего типа:

```
var A1: MyArray1;
```

Другим вариантом создания массива является одновременное объявление как переменной, так и описания массива:

```
var A1: array [1..100] of Integer;
```

Если же необходимый размер массива заранее неизвестен, то можно использовать динамические массивы. Они могут определяться как с предварительным объявлением типа, так и без такового – достаточно создать переменную и в качестве ее типа указать массив:

```
var DynArray: array of integer;
```

Однако для того, чтобы приступить к использованию динамического массива, вначале требуется выделить для него место в памяти. Делается это при помощи процедуры SetLength, указав имя динамического массива и необходимый размер памяти:

```
SetLength(DynArray, 10);
```

Преимуществом динамического массива является то, что по ходу выполнения программы, количество выделяемой под массив память можно изменять.

Для обращения к конкретному элементу массива используется индекс (или индексы) элемента в массиве. Так, первый элемент в массиве типа MyArray1 (array [1..100]) имеет индекс, равный 1, а последний – 100. Соответственно, чтобы обратиться к первому элементу массива, скажем для того, чтобы присвоить ему значение, используются записи подобного типа:

```
A1[1] := 10;
```

Здесь мы присвоили элементу массива A1 с индексом 1 значение 10. Считывание данных производится аналогичным образом:

```
x := A1[1];
```

В данном случае переменной x будет присвоено значение 1-го элемента массива A1. При этом важно, чтобы было соблюдено 2 условия: во-первых, тот элемент массива, к которому идет обращение, должен существовать. Т.е. если обратиться к 11-му элементу массива, состоящего из 10 элементом, то произойдет ошибка доступа к памяти. Во-вторых, тип присваиваемых данных должен совпадать. Иначе говоря, если массив определен как целочисленный, то только целочисленные значения можно ему присваивать. При обратной же ситуации (считывания данных) правила несколько мягче, поскольку целое число можно присвоить переменой как целочисленного, так и вещественного типа.

В то же время, важной (и крайне полезной) особенностью массивов является то, что они могут хранить в себе данные любого типа, включая как любой простой, так и

структурный тип, в том числе записи, объекты и другие массивы. Собственно говоря, двумерный массив можно охарактеризовать как одномерный массив, каждый элемент которого так же является одномерным массивом. С этой точки зрения становится понятным, почему для двумерных массивов допускается использовать 2 типа объявления и доступа к элементам, причем оба они полностью взаимозаменяемы и аналогичны по смыслу:

```
var A1: array [1..10, 1..10] of integer;
var A2: array [1..10] of array [1..10] of integer;
...
A1[1][3]:=5;
A2[1][3]:=5;
A1[1,3]:=5;
A2[1,3]:=5;
```

В данном примере оба объявленных массива (A1 и A2) полностью идентичны. Точно так же идентичны и обращения к элементам массивов – в обоих случаях можно использовать как синтаксис с отдельными значениями индексов, так и с индексами, перечисляемыми через запятую. Многомерными могут быть не только статические массивы, но и динамические, при этом используется 2-й вариант объявления. Например, двумерный динамический массив вещественных чисел объявляется следующим образом:

```
var DynArray: array of array of real;
```

При выделении памяти под такой массив так же следует учитывать его размерность. Например, если требуется массив размером 10 на 20 элементов, то следует использовать процедуру SetLength со следующими параметрами:

```
SetLength(DynArray, 10, 20);
```

В завершение знакомства с массивами нам осталось рассмотреть обещанный ранее пример использования циклов для заполнения массива. Допустим, у нас имеется простой массив-вектор для 10 целых чисел:

```
var MyArray: array [1..10] of integer;
```

Таким образом, если нам надо заполнить его значениями, следующими по порядку (скажем, числа от 10 до 100 с шагом 10), то вместо того, чтобы последовательно присваивать каждому элементу массива свое значение, можно использовать следующий цикл:

```
for i := 1 to 10 do MyArray[i] := i * 10;
```

Здесь переменная i, являющаяся счетчиком цикла, при каждой его итерации последовательно увеличивается на 1. В результате каждый элемент массива MyArray получает значение этой переменной, умноженной на 10, т.е. 10, 20, 30, 40 и т.д. – чего нам и требовалось. Чтобы убедиться в этом, а заодно продемонстрировать цикл для считывания данных из массива, обратимся к примеру, приведенному в листинге 5.1 (на CD этот пример находится в папке Demo\Part1\ArrayFor):

Листинг 5.1. Запись и считывание данных массивов в цикле

```
program arrayfor;
{$APPTYPE CONSOLE}
var
 MyArray: array [1..10] of integer;
```

```
 i :integer;
begin
 for i := 1 to 10 do MyArray[i] := i * 10; // заполнение массива
 for i := 1 to 10 do writeln(MyArray[i]);  // вывод массива
 readln; // ожидание ввода для предотвращения закрытия окна
end.
```

Относительно массивов можно сделать еще одно замечание: один уже известный нам тип данных – строки (string) можно рассматривать как массив символов (char). При этом никаких предварительных преобразований не требуется: если у нас уже есть строка, то всегда можно обратиться к отдельному ее символу как к элементу массива:

```
var
 s: string;
 c: char;
...
s := 'Москва';
c := s[1];
```

В данном случае переменная c получит в качестве своего значения букву «М», т.е. первый символ строки.

Множества

Иногда бывает удобным ограничить возможные значения переменной только частью значений из множества всех значений, допускаемых ее типом. Допустим, нам нужна переменная типа Char, которая может принимать значения только из строчных латинских символов. В таком случае нам пригодится такое средство, как подмножество, в данном случае – подмножество символов от a до z. Определить его можно так:

```
type SmLetter = 'a'..'z';
```

Таким образом, мы получили новый тип данных – SmLetter, который является «урезанным» вариантом типа Char. При использовании такого типа данных, переменные типа SmLetter не смогут принимать значения, выходящие за пределы указанного диапазона:

```
var a: SmLetter;
...
a := 'b'; // здесь все правильно, т.к. малая b входит в подмножество a..z
a := 'B'; // ошибка! Прописная B не входит в подмножество a..z
```

Практическими кандидатами в подмножества могут быть переменные, предназначенные для хранения отдельных фрагментов дат (скажем, от 1 до 12 для месяцев и от 1 до 31 – для дней месяца), для проверки значений, которые программы получает в результате ввода пользователя с клавиатуры, и т.д. Следует лишь учитывать, что подмножества, по понятным причинам, могут принадлежать только к ординарным типам данных – целым числам или символам.

Развитием подмножеств являются множества, или наборы (sets). Они, подобно подмножествам, могут быть выделены в отдельные типы данных, однако допустимые диапазоны их значений определяются несколько иначе. Прежде всего, для определения множества используется ключевое слово set:

```
Set of <тип элементов>
```

Если провести аналогию с предыдущим примером, то диапазон из строчных латинских символов можно определить следующим образом:

```
type Letters = set of Char;
var a: Letters;
...
a := ['a'..'z'];
```

Таким образом, определение множества состоит из двух этапов: вначале определяется тип, на основании которого строится подмножество (Letters), затем объявляется переменная этого типа (a), и уже к переменной применяется диапазон. Преимущество здесь состоит в том, что, во-первых, по ходу программы можно менять допустимые диапазоны значений, а во-вторых, сами диапазоны определяются гораздо более гибко. В частности, они могут содержать в себе как ряды отдельных значений, так и подмножества, или их сочетания в любой последовательности. Например, если нам надо выделить только некоторые символы, скажем, прописные от A до K, а так же цифры 1, 3, 4, 5 и 9, то определение группы получится следующим:

```
type Letters = set of Char;
var a: Letters;
...
a := ['1', '3'..'5', '9', 'A'..'K'];
```

Для проверки, является ли то или иное значение членом множества, используется операция in. Например, чтобы проверить, относится ли введенный пользователем символ (обозначим его как переменную «c») к множеству a, достаточно такого выражения:

```
if c in a then ...
```

Чтобы продемонстрировать работу множеств и операции in, обратимся к примеру, приведенному в листинге 5.2 (Demo\Part1\Ranges).

Листинг 5.2. Операция in и подмножества

```
program rangeset;
{$APPTYPE CONSOLE}
type Letters = set of Char;
var
 a: Letters;
 c: Char;
begin
 a := ['1', '3'..'5', '9', 'A'..'K']; // определение группы
 readln(c); // считывание ввода пользователя
 if c in a then writeln('Input is Ok!') else writeln('Input is Wrong!');
 readln; // ожидание ввода для предотвращения закрытия окна
end.
```

Еще одной разновидностью множества является перечисление. Использование перечислений призвано, прежде всего, улучшить читаемость (восприятие) кода программы. Допустим, в программе требуется неоднократно определять текущую раскладку клавиатуры, причем предусмотрено 3 состояния – русская, английская и другая. Разумеется, каждому состоянию можно назначить цифру, скажем, 1, 2 и 3 соответст-

венно, однако по ходу написания программы всякий раз придется вспоминать, что означает та или иная цифра:

```
if KeyLang = 2 then ... // по прошествии месяца вспомните, что тут значит 2!
```

На помощь здесь приходят перечисляемые типы. Они определяются следующим образом:

```
<Имя типа> = (<название значения1>, ... < название значенияN>);
```

Например, в варианте для трех значений раскладки клавиатуры мы получим:

```
type TKeyLang = (klRussian, klEnglish, klOther);
```

Согласитесь, что klEnglish куда понятнее, чем просто цифра «2». Более того, на основании заданного таким образом перечисления можно создать тип-множество. Для этого после того, как исходный тип с диапазоном определен, определяют имя типа на его основе:

```
type TKeyLang = (klRussian, klEnglish, klOther);
TKeyLangs = set of TKeyLang;
```

ВНИМАНИЕ

Обратите внимание на то, что имена типов начинаются с буквы «T». Хотя это и не является требованием языка, однако начинать названия сложных типов с этой буквы (от слова Type – тип) де-факто является стандартом.

Таким образом, в программе можно будет использовать уже не исходный тип, в котором определен состав множества, а новый тип, являющийся его производным:

```
var KeyLang: TKeyLangs;
...
if KeyLang = klEnglish then ;
```

ПРИМЕЧАНИЕ

Объявление типов-перечислений в 2 этапа является наиболее распространенной практикой. При этом исходный тип определяется именем в единственном числе (TKeyLang), а производный – во множественном, с «s» на конце (TKeyLangs).

Записи

Еще одной полезной разновидностью пользовательских видов данных являются записи (records). Запись, подобно массиву, может хранить в себе целый набор данных, но при этом они не должны принадлежать к одному типу. Иначе говоря, если массивы предназначены для хранения каких-либо рядов данных, то записи – для объединения разнородной информации под общим именем. Например, запись – просто незаменимый тип данных для хранения такой информации, как адреса или информация о персоне. И в том и в другом случае под общим именем должны быть собраны данные разного типа – строковые, численные и т.д. Например, для адреса это почтовый индекс, названия города и улицы, а так же номера дома и квартиры. Всю эту разнотипную информацию можно собрать под общей вывеской одной записи.

Объявление записи начинается с ключевого слова record, за которым следует перечисление всех входящих в нее элементов, называемых полями записи, и завершается ключевым словом end:

```
<название записи> record
  <имя поля 1> : <тип поля 1>;
  ...
  <имя поля N> : <тип поля N>;
end;
```

Применительно к почтовому адресу определение записи может выглядеть таким образом:

```
type TAddress = record
  PostIndex : integer;
  City      : string;
  Street    : string;
  HouseNr   : integer;
  FlatNr    : integer;
end;
```

Обращение к отдельным полям записей производится при помощи точечной нотации, т.е. когда после имени переменной, являющейся записью, ставят точку, и сразу после точки указывают название того поля, к которому надо обратиться. Например, мы можем создать переменную типа TAddress и назвать ее, скажем, MyAddr, после чего заполнить те или иные ее поля, воспользовавшись для доступа к ним точкой:

```
var MyAddr: TAddress;
...
MyAddr.PostIndex := 119071;
MyAddr.City := 'Москва';
```

В том случае, если полем записи является другая запись, то точка используется дважды. Так, если у нас определена еще одна запись, для хранения информации о персоне, то одним из ее полей наверняка окажется адрес, а у нас уже есть подходящий тип данных для этого, так что можно использовать его в качестве поля:

```
type TPerson = record
  Name      : string;
  Phone     : string;
  Address   : TAddress;
end;
```

Таким образом, все поля типа TAddress будут принадлежать также и к типу TPerson, но обращаться к ним надо не напрямую, а через поле записи TPerson соответствующего типа, т.е. через Address. Такие поля называются составными:

```
var Anybody: TPerson;
...
Anybody.Name := 'Вася Иванов';
Anybody.Address.PostIndex := 119071;
Anybody.Address.City := 'Москва';
```

В то же время, можно поступить и по другому: создать переменную типа TAddress, заполнить ее значениями, а затем присвоить соответствующему полю переменной типа TPerson. В примере, приведенном в листинге 5.3, продемонстрированы оба способа работы с составными полями, а так же продемонстрировано, что со всеми полями можно делать то же самое, что и с отдельными переменными того же типа.

Листинг 5.3. Записи

```
program recdemo;
{$APPTYPE CONSOLE}
type
 TAddress = record
  PostIndex : integer;
  City      : string;
  Street    : string;
  HouseNr   : integer;
  FlatNr    : integer;
 end;
 TPerson = record
  Name      : string;
  Phone     : string;
  Address   : TAddress;
 end;
var
 Anybody: TPerson;
 Address: TAddress;
begin
 write('Name: ');
 readln(Anybody.Name);
 write('Phone: ');
 readln(Anybody.Phone);
 write('Postal Index: ');
 readln(Address.PostIndex);
 write('City: ');
 readln(Address.City);
 write('Street: ');
 readln(Address.Street);
 write('House number: ');
 readln(Address.HouseNr);
 write('Flat number: ');
 readln(Address.FlatNr);
 Anybody.Address:=Address;
 writeln(Anybody.Name);
 writeln(Anybody.Phone);
 writeln(Anybody.Address.PostIndex);
 writeln(Anybody.Address.City);
 writeln(Anybody.Address.Street);
 writeln(Anybody.Address.HouseNr);
 writeln(Anybody.Address.FlatNr);
 readln;
end.
```

Приведенная в листинге программа последовательно предлагает пользователю ввести свойства – сначала для записи о персоне, а затем – для адреса, после чего полю адреса персоны присваивается значение записи-адреса. После этого все поля после-

довательно выводятся на экран. Исходный код программы можно найти в папке Demo\Part1\Records (файл recdemo.dpr).

В завершение темы записей рассмотрим еще одну их особенность, характерную для Object Pascal, а именно – возможность сочетать в одном типе записи с разными полями. Называются такие записи вариантными и объявляются так же как и обычные, за исключением того, что содержат дополнительную часть, начинающуюся с ключевого слова case:

```
<название записи> record
  [<имя поля 1> : <тип поля 1>;
  ...
  <имя поля N> : <тип поля N>;]
  case [<Признак> :] <Тип признака> of
  <Вариант 1> : (<Поля для варианта 1>);
  ...
  <Вариант N> : (<Поля для варианта N>);
end;
```

В качестве примера можно привести такой случай, когда необходимо создать тип записи «служащий», в котором требуется имя и размер оплаты. Дело в том, что эти самые служащие могут быть как принятыми на постоянную работу, и имеет помесячный оклад, так и временные, для которых применяется почасовая оплата. Важно отметить, что одновременно может быть только один вид оклада. Соответственно, мы можем использовать вариантный тип записи, в качестве признака в которой будет выступать булево поле «Salaried» («на окладе»):

```
type
  TEmployee = record
  Name: string;
  JobTitle: string;
  case Salaried: Boolean of
    true: (Salary: Currency);
    false: (Hourly: Currency);
end;
```

Здесь, в зависимости от того, будет ли значением поля Salaried той или иной переменной типа TEmployee ложь или истина, у нее будет либо поле Salary, либо Hourly. Пример использования подобной вариантной записи вы можете посмотреть в файле varrec.dpr.

Специальные типы данных

Помимо уже рассмотренных типов данных, простых и пользовательских, в Object Pascal имеется ряд иных, специализированных типов. Например, для времени используют тип TDateTime. В принципе, этот тип алогичен вещественному типу Double, однако для работы с такими данными, как временные диапазоны, удобнее использовать специальный тип TDateTime.

Поскольку этот тип является, фактически, вещественным числом, то данные хранятся в нем следующим образом: целая часть числа определяет дату, за которую берется количество дней, прошедших с 31 декабря 1899 года, а дробная определяет время в

миллисекундах, прошедших с начала текущего дня. Преимуществом же типа TDateTime является то, что для него предусмотрен целый набор готовых функций, позволяющих работать с датами и временем. Их список приведен в таблице 5.1.

Таблица 5.1. Функции для работы с датой и временем

Функция	Описание
Now	Возвращает текущие дату и время
Date	Возвращает текущую дату (целую часть TDateTime)
Time	Возвращает текущее время (дробную часть TDateTime)
DateTimeToStr	Преобразует дату и время в строку на основе системных настроек
DateTimeToString	Копирует дату и время в указанную строковую переменную
DateToStr	Преобразует дату в строку
TimeToStr	Преобразует время в строку
FormatDateTime	Преобразует дату и время в указанный формат
StrToDateTime	Преобразует строку, содержащую написанную надлежащим способом дату и время, в переменную типа TDateTime
StrToDate	Преобразует строку в дату в формате TDateTime
StrToTime	Преобразует строку во время в формате TDateTime
DayOfWeek	Возвращает номер дня недели (от 1 до 7) для указанной даты. Учитывайте, что 1-й день недели – воскресенье
DecodeDate	Раскладывает значение типа TDateTime на 3 целых, представляющих собой год, месяц и день месяца
DecodeTime	Раскладывает значение типа TDateTime на 4 целых, представляющих собой часы, минуты, секунды и миллисекунды
EncodeDate	Объединяет 3 целых, представляющих собой год, месяц и день, в одно значение типа TDateTime
EncodeTime	Объединяет 4 целых, представляющих собой часы, минуты, секунды и миллисекунды? в одно значение типа TDateTime

Типичный пример использования функций для работы с датами, может выглядеть примерно таким образом:

```
var
 today, yesterday: TDateTime;
 s: string;
...
today := Now();
yesterday := today - 1;
s := TateToStr(yesterday);
```

Здесь переменной s будет назначено значение, соответствующее вчерашнему дню в формате, принятому в системе (например, «16.07.2005»). Более полный пример работы с датами вы можете посмотреть в Demo\Part1\Dates.

Кроме дат, рассмотрим еще один тип данных – файлы (files). Файлы представляет собой некую последовательность однотипных элементов, размещенных на внешнем носителе, не находящемся в оперативной памяти ПК. В типичном случае таким носителем является жесткий диск. Этот тип данных можно охарактеризовать как одно-

мерный массив без указания размера. Чтобы производить те или иные операции над такими данными, используют специальную переменную файлового типа. Причем в зависимости от того, с какими данными предстоит работать, задают тот или иной тип файла. Например, для работы с текстовыми файлами используют специальный тип TextFile, а если файл содержит в себе ряд чисел, то определяют, какой тип чисел в нем используется:

```
var
 f1: TextFile; // текстовый файл
 f2: File of integer; // файл с целыми числами
 f3: File of double; // файл с вещественными числами
```

Если же файл нетипизированный, (например, бинарный), то используют тип File без каких-либо дополнений:

```
var f4: File; // двоичный файл или файл заранее неизвестного типа
```

Работа с файловыми переменными имеет ряд особенностей. Прежде всего, просто объявить переменную файлового типа мало – необходимо так же связать ее с каким-либо конкретным файлом на диске. После этого файл следует открыть, при этом указывают режим файла – его можно открывать для чтения или для записи, или для чтения и записи одновременно.

ПРИМЕЧАНИЕ

> Саму файловую переменную называют дескриптором файла, т.е. фактически она лишь указывает программе на место в памяти, через которое обеспечивается доступ к файлу средствами операционной системы.

Для работы с файлами так же предусмотрен целый ряд процедур и функций. Среди них можно отметить уже знакомые нам read/readln и write/writeln. Чтобы эти процедуры работали с файлами, в качестве первого параметра указывают имя файловой переменной (дескриптор файла):

```
writeln(f, 'Текст для записи в файл');
```

Но прежде, чем производить запись в файл, или начать считывать из него данные, как уже говорилось, надо связать переменную с файлом (назначить дескриптор) и открыть его, попутно назначив режим доступа. Назначение дескриптора файлу производится с помощью процедуры AssignFile, например

```
AssignFile(f, 'c:\file.txt');
```

Что касается открытия файла, то тут дела обстоят несколько сложнее, поскольку следует учитывать тип файла и режим доступа. Так, применительно к текстовым файлам, используют процедуры Reset, Rewrite и Append, открывающие файл на чтение, перезапись и добавление (запись в конец файла), соответственно.

Наконец, следует отметить, что после того, как операции с файлом произведены, его необходимо закрыть. Для закрытия файла используется процедура CloseFile. Таким образом, вариант использование всех этих процедур для чтения и записи файла в общем случае, выглядит таким образом, как показано в листинге 5.4:

Листинг 5.4. Запись и чтение в файлах

```
program readwrite;
{$APPTYPE CONSOLE}
```

```
uses
 SysUtils;
var
 f: TextFile;
 s: string;
begin
 AssignFile(f, 'c:\test.txt'); // назначаем дескриптор файлу text.txt
 Rewrite(f); // открываем файл на запись
 writeln(f, s); // производим запись в файл
 CloseFile(f); // закрываем файл
 Reset(f); // открываем файл на чтение
 readln(f, s); // считываем данные из файла
 CloseFile(f); // закрываем файл
end;
```

Еще один пример работы с файлами можно посмотреть в Demo\Part1\Files. Вместе с тем, на практике файловые типы данных не часто используются при современном программировании в среде Delphi, поскольку VCL предлагает ряд более удобных и изящных методов для хранения данных на диске, начиная от методов отдельных классов и заканчивая потоками и базами данных.

Совместимость и преобразование типов

При рассмотрении простых типов мы уже поднимали вопрос их совместимости и преобразования друг в друга. Теперь настала пора рассмотреть этот аспект более внимательно. В частности, если целое число без проблем приводится к вещественному, то как быть в том случае, если требуется обратное преобразование? Выход в данной ситуации состоит в использовании специальных функций преобразования типов. Так, для преобразования вещественного числа в целое используются функции Round и Trunc. Их отличие состоит в том, что первая округляет значение до целого, опираясь на стандартные математические правила, а вторая – просто отбрасывает дробную часть числа. Отметим, что если надо просто отбросить дробную часть числа, оставив тип данных без изменений, то следует использовать другую функцию – Int. Примеры их использования показаны ниже:

```
var
 i: integer;
 r: real;
...
r := 5.75;
i := Round(r); // i получит значение 6
i := Trunc(r); // i получит значение 5
r := Int(r); // r получит значение 5.0
```

Куда большее количество функций предусмотрено для преобразования числовых типов в строковые и наоборот. С некоторыми из них, предназначенных для дат, мы уже знакомы (см. таблицу 5.1). Другие же представлены в таблице 5.2.

Таблица 5.2. Функции для преобразования чисел в строки и наоборот

Функция	Описание
IntToStr	Преобразует целое число в строку

StrToInt	Преобразует строку в целое число, в случае невозможности преобразования вызывает ошибку
StrToIntDef	Преобразует строку в целое число, в случае невозможности преобразования возвращает число, указанное в качестве второго аргумента
FloatToStr	Преобразует вещественное число в строку
FloatToStrF	Преобразует вещественное число в строку на основе указанного формата
StrToFloat	Преобразует строку в вещественное число, в случае невозможности преобразования вызывает ошибку
StrToFloatDef	Преобразует строку в вещественное число, в случае невозможности преобразования возвращает число, указанное в качестве второго аргумента
CurrToStr	Преобразует число типа Currency в строку
CurrToStrF	Преобразует число типа Currency в строку на основе указанного формата
StrToCurr	Преобразует строку в число типа Currency, в случае невозможности преобразования вызывает ошибку
StrToCurrDef	Преобразует строку в число типа Currency, в случае невозможности преобразования возвращает число, указанное в качестве второго аргумента

В качестве формата в функциях типа FloatToStrF подразумевают один из предопределенных вариантов форматирования, а так же количество знаков после запятой и общее количество. Например, для того, чтобы оставить не более 2 знаков после запятой, можно использовать следующее выражение:

```
str := FloatToStrF(x, ffGeneral, 10, 2);
```

Здесь ffGeneral является указанием на формат вывода, 10 определяет максимально возможное число знаков в числе вообще, а 2 – предельно допустимое число знаков после запятой. Помимо ffGeneral, определяющего наиболее обобщенный формат представления чисел, имеются и другие:

- ffExponent – формат с экспонентой (например, 1.45E10);
- ffFixed – фиксированный формат (например, 145000.01);
- ffNumber – «прописной» формат (например, 1,450,000.0);
- ffCurrency – валютный формат (например, 145 000,00р).

Таким образом, привести то или иное число к строке нужного формата, оказывается достаточно просто – важно лишь определиться, что нужно получить. А еще одна функция – Chr – позволяет преобразовывать маленькие числа (до 255) в символы, т.е. фактически, из типа Byte делает Char.

Что касается обратных преобразований (строк в числа), то тут следует соблюдать определенную осторожность, поскольку далеко не всякая строка может стать числом. В первую очередь это касается преобразований вещественных числе, поскольку нередко вместо отделяющей мантиссу точки может оказаться запятая. В случаях, когда преобразование строки в число невозможно, возникает ошибка выполнения программы. Ее можно обрабатывать самостоятельно (об обработке ошибок будет рассказано во второй части книги), или же поручить это дело функции – в таком случае следует использовать функции с суффиксом Def (StrToIntDef, StrToFloatDef и StrToCurrDef). В качестве второго аргумента они принимают значение, которое следует использовать в том случае, если преобразование невозможно:

```
x := StrToIntDef(str, -1);
```

В данном случае, если в строке str не удастся распознать число, переменной x будет присвоено значение -1.

При всем многообразии типов данных, в Object Pascal существует тип еще один тип, который не имеет типа - вариантный. Называется он Variant. Переменные вариантного типа могут принимать значения любого простого типа, а так же некоторых специальных. В типичном случае использование вариантного типа может выглядеть так:

```
var
 v: variant;
...
v := 5;
v := 'Строковое значение';
v := 10.54;
```

Этот пример иллюстрирует, как одной т той же переменной последовательно присваиваются данные 3 разных типов – целочисленного, строкового и вещественного. При этом никакой ошибки не возникает. Однако вариантные данные обрабатываются гораздо медленнее, чем типизированные – практически так же медленно, как программы на языке BASIC. (кстати, в классическом BASIC как раз только вариантные данные и были). Кроме того, использование нетипизированных данных вообще, а в строго типизированном языке – особенно, чревато непредсказуемым поведением программы и прочими ошибками. Поэтому оставим этот тип для внутреннего использования Delphi – в VCL он применяется для работы с OLE и базами данных.

Указатели

Указатели (pointers) – это такой тип переменных, которые хранят адрес в памяти компьютера, по которому расположена другая переменная. Фактически, указатель не содержит значение, а ссылается на него.

Указатели можно задать двумя принципиально разными способами. Во-первых, можно использовать специальный тип – Pointer. При этом будет создан нетипизированный указатель, под который всякий раз надо будет принудительно выделять память, используя функцию GetMem. Другой, как правило более предпочтительный способ, состоит в том, что сразу же создается указатель нужного типа. Делается это при помощи символа «^», предшествующего названию типа:

```
var P: ^integer;
```

Здесь мы определили указатель P, являющийся указателем на переменную целочисленного типа.

После того, как указатель создан, можно связать его с переменной подходящего типа, используя операцию *@*:

```
var
 P: ^integer;
 x: integer;
...
P := @x;
```

Теперь к переменной x можно обращаться как непосредственно, так и через ее указатель. В случае обращения через указатель так же используют символ «^»:

```
x := 10;
P^ := 10;
```

В обоих случаях переменная x получит значение 10.

Другим вариантом использования указателя, помимо связывания с существующей переменной, является выделение для него памяти и дальнейшая работа как с ссылкой на некую абстрактную переменную (фактически, непосредственно на область в памяти, выделенную для хранения данных). В таком случае, код будет выглядеть так:

```
var
 P: ^integer;
...
New(P); // выделение памяти, необходимой для хранения данных типа Integer
P^ := 10; // занесение данных в выделенный блок памяти
Dispose(P); // освобождение памяти
```

При работе с указателями таким методом следует самостоятельно заботиться о выделении и очистке памяти под данные.

В целом же указатели, как правило, используются для взаимодействия с системными функциями операционной системы (в частности, с Windows API), для взаимодействия со сторонними программами и динамически подключаемыми библиотеками.

Объекты

Самые сложные и интересные типы данных – это объекты. В современных версиях Delphi объекты бывают 3 основных типов: собственно объекты, а так же классы и интерфейсы. Тип объекта (object) достался Delphi от предшественника – языка Pascal with Objects, и в настоящее время практически не используется. Основным типом объектных данных в современных программах является класс (class). Что касается интерфейсов (interface), то они являются разновидностью классов, и предназначены для взаимодействия с системными объектными функциями Windows.

Тема объектов достаточно обширна, поскольку является основой для парадигмы объектно-ориентированного программирования. ООП в Object Pascal рассматривается во второй части настоящего издания.

Процедуры и функции

Изучив основные «кирпичики», из которых составляются программные инструкции, а именно – переменные и операторы, мы можем приступить к исследованию вопросов их эффективного расположения в теле программы. Для этих целей рассмотрим вопрос использования подпрограмм.

О подпрограммах в Object Pascal

Важной составной частью программирования в Object Pascal является использование подпрограмм – специальным образом оформленных и логически законченных блоков инструкций. Подпрограмму можно вызывать любое число раз из других мест программы, или из других подпрограмм. Таким образом, использование подпрограмм позволяет сделать исходный код более стройным и наглядным.

Структура подпрограммы похожа на программу в миниатюре: она содержит заголовок, блок объявления переменных и блок инструкций. Из отличий можно выделить лишь невозможность подключать модули (блок uses), а так же ограничения на объявления типов данных: если локальные простые и даже составные типы в подпрограммах вполне допустимы, то более сложные типы – объекты, классы и интерфейсы, локальными быть не могут, а потому в подпрограммах их объявлять нельзя.

Использование подпрограммы состоит из 2 этапов: сначала подпрограмму описывают, а затем, уже в блоке инструкций программы, вызывают. Отметим, что в библиотеке Delphi имеется описание тысяч готовых подпрограмм, описывать которые, разумеется, уже не надо. А их вызовом мы уже неоднократно занимались – достаточно взглянуть на любой пример, где мы встречали инструкции, подобные таким:

```
write('Hello, world!');
readln;
```

Здесь и write, и readln – стандартные подпрограммы Object Pascal. Таким образом, с вызовом подпрограмм мы уже знакомы. Осталось узнать, как создавать собственные, или пользовательские, подпрограммы. Но прежде отметим, что все подпрограммы делятся на 2 лагеря: процедуры и функции. Мы уже использовали эти термины, и даже давали им описание, однако повторимся: процедуры – это такие подпрограммы, которые выполняют предназначенное действие и возвращают выполнение в точку вызова. Функции в целом аналогичны процедурам, за тем исключением, что они еще и возвращают результат своего выполнения. Результатом работы функции могут быть данные любого типа, включая объекты.

Вместе с тем, значение, возвращаемое функцией, можно проигнорировать, в таком случае она ничем не будет отличаться от процедуры. Разумеется, при этом функция все-таки должна выполнить какое-либо действие, сказывающееся на выполнении программы, иначе она потеряет всякий смысл. С другой стороны, процедуры могут возвращать значения через свои параметры – например, как это делает DecodeDate. Таким образом, различия между процедурами и функциями в современном программировании весьма призрачны.

Как процедурам, так и функциям могут передаваться данные для обработки. Делается это при помощи списка параметров. Список параметров в описании подпрограммы и список аргументов, указываемых при ее вызове должен совпадать. Иначе говоря, если в описании определено 2 параметра типа Integer, то, вызывая такую подпрограмму, в качестве аргументов так же следует указать именно 2 аргумента и именно типа Integer или совместимого (скажем, Word или Int64).

ПРИМЕЧАНИЕ

> На самом деле, Object Pascal позволяет довольно гибко обращаться с аргументами, для чего имеются различные методы, включая «перегружаемые» функции, значения параметров по умолчанию и т.д. Тем не менее, в типичном случае, количество, тип, и порядок перечисления аргументов при объявлении и при вызове процедуры или функции, должны совпадать.

Любые подпрограммы выполняются до тех пор, пока не будет выполнена последняя инструкция в блоке подпрограммы, или пока в ее теле не встретится специальная процедура exit. Процедура exit досрочно прерывает выполнение подпрограммы и возвращает управление инструкции, следующей за вызовом данной подпрограммы.

Процедуры

Итак, начнем исследование подпрограммы с процедур. Как уже было отмечено, процедуру надо описать. Описание процедуры состоит из заголовка и тела процедуры.

Заголовок состоит из ключевого слова procedure, за которым следует имя процедуры и, при необходимости, список параметров, заключенных в круглые скобки:

```
procedure <Имя> [(параметры)];
```

Вслед за заголовком может следовать блок объявления локальных меток, типов и переменных. Локальными они называются потому, что предназначены исключительно для этой процедуры.

ПРИМЕЧАНИЕ

> Вопросы локальных и глобальных переменных, и вообще видимости в программах, будет рассмотрен позже в этой главе.

После заголовочной части следует тело процедуры, заключаемое в begin и end. Таким образом, исходный код процедуры может выглядеть примерно таким образом:

```
procedure TriplePrint(str: string);
var
 i: integer;
begin
 for i := 1 to 3 do begin
   writeln('"'+str+'"');
 end; // конец цикла for
end; // конец процедуры TriplePrint
```

Здесь мы определили процедуру TriplePrint, которая будет трижды выводить переданную ей в качестве аргумента строку, заключенную в двойные кавычки. Как видно, данная процедура имеет все составные части: ключевое слово procedure, имя, список параметров (в данном случае он всего один – строковая переменная str), блок объявления собственных переменных (целочисленная переменная i), и собственное тело, состоящее из оператора цикла for.

Для использования данной процедуры в любом месте программы достаточно написать инструкцию вызова процедуры, состоящую из имени процедуры и списка аргументов, например:

```
TriplePrint('Hello!!!');
```

Отметим так же, что рассмотренная нами процедура сама содержит вызов другой процедуры – writeln. Процедуры могут быть встроенными. Иначе говоря, объявление одной процедуры можно помещать в заголовочную часть другой. Например, наша процедура TriplePrint может иметь вспомогательную процедуру, которая будет «подготавливать» строку к выводу. Для этого перед объявлением переменной i, разместим объявление еще одной процедуры. Назовем ее PrepareStr:

```
procedure PrepareStr;
begin
 str := '"'+str+'"';
end;
```

Отметим, что переменная str, хотя и не передается этой процедуре в качестве параметра, тем не менее может в ней использоваться, поскольку данная процедура является составной частью процедуры TriplePrint, внутри которой данная переменная доступна для использования.

Таким образом, мы получаем две процедуры, одна из которых (TriplePrint) может использоваться во всей программе, а другая (PrepareStr) – только внутри процедуры TriplePrint. Чтобы преимущество использования процедур было очевидно, рассмотрим их на примере программы, которая будет использовать ее неоднократно, для чего обратимся к листингу 6.1 (см. так же пример в Demo\Part1\Procs).

Листинг 6.1. Использование процедур

```
program procs;
{$APPTYPE CONSOLE}
procedure TriplePrint(str: string);
 procedure PrepareStr;
 begin
  str := '"'+str+'"';
 end;
var
 i: integer;
begin
 PrepareStr;
 for i := 1 to 3 do begin
   writeln(str);
 end;
end; // конец процедуры TriplePrint
begin // начало тела основной программы
 TriplePrint('Hello!!!'); // первый вызов TriplePrint
 TriplePrint('How are you???'); // 2-й вызов
 TriplePrint('Bye...'); // 3-й
 readln;
end.
```

Очевидно, что если бы не процедура, то нам трижды пришлось бы писать цикл, свой для каждого слова. Таким образом, процедуры позволяют использовать единожды написанный код многократно, что существенно облегчает написание программ.

Функции

Подобно процедурам, описание функции состоит из заголовка и тела. Однако описание заголовка имеет 2 отличия: прежде всего, для функций используется ключевое слово function. Кроме того, поскольку функции всегда возвращают результат, завершается строка заголовка типом возвращаемого значения. Таким образом, для объявления функции мы получаем следующий синтаксис:

```
function <Имя> [(параметры)] : <тип результата>;
```

Возвращаемое значение может быть любого типа, кроме файлового. Что касается дальнейшего описания функции, то оно полностью аналогично таковому для процедур. Единственным дополнением является то, что в теле функции обязательно долж-

на присутствовать хотя бы одна операция присваивания, в левой части которой должно быть либо имя функции, либо ключевое слово result. Именно это выражение и определяет возвращаемое функцией значение.

Рассмотрим пример функции, которая будет возвращать куб числа, переданного ей в качестве аргумента:

```
function cube(value: integer) : integer;
  result := value * value * value;
}
```

Здесь определена функция, имеющая параметр value типа целого числа, которое она возводит в третью степень путем троекратного умножения, и результат присваивается специальной переменной result. Таким образом, чтобы в любом месте программы вычислить значение числа в 3-й степени, достаточно написать такое выражение:

```
x = cube(3);
```

В результате выполнения этого выражения переменной x будет присвоено значение 27. Данный пример иллюстрирует использование функций в классическом случае – для явного вычисления значения переменной. Однако функции могут использоваться в выражениях и напрямую. Например, можно поставить вызов функции cube в какомлибо месте арифметического выражения подобно обычной переменной:

```
x := a + cube(b) / 2;
```

Подобно процедурам, функции так же могут быть встроенными. Кроме того, функции могут включать в себя не только локальные функции, но и процедуры. Впрочем, верно и обратное – в процедурах могут использоваться локальные функции. Например, в той же процедуре TriplePrint можно было бы использовать не процедуру, а функцию PrepareStr, которая принимала бы строку и возвращала ее же в кавычках:

```
procedure TriplePrint(str: string);
 function PrepareStr(s: string) : string;
 begin
  result := '"'+s+'"';
 end;
var
 i: integer;
begin
 for i := 1 to 3 do begin
   writeln(PrepareStr(str)); // функция использована как переменная
 end;
end;
```

Как уже отмечалось, помимо специальной переменной result, в функциях можно использовать другую автоматически объявляемую переменную, имя которой соответствует имени функции. Так, для функции cube имя переменной также будет cube:

```
function cube(value: integer) : integer;
  cube := value * value * value;
}
```

В данном случае оба варианта будут вести себя полностью аналогично. Различия проявляются лишь в том случае, если использовать такую переменную в выражениях в теле функции. В подобных случаях следует использовать именно переменную re-

result, а не имя функции, поскольку использ0овании имени функции в выражении внутри самой функции приведет к ее рекурсивному вызову.

Рекурсия

Таким образом мы подошли к теме рекурсии – вызову подпрограммы из самой себя. Это не является ошибкой, более того, целый ряд алгоритмов решить без рекурсии вообще было бы затруднительно.

Рассмотрим вопрос рекурсии на следующем примере:

```
function recfunc(x: integer) : integer
begin
  dec(x); // функция декремента, уменьшает целое на 1
  if x > 5 then x := recfunc(x);
  result := 0; // возвращаемое значение тут не используется
end;
```

Здесь мы объявили функцию recfunc, принимающую один аргумент, и вызывающую саму себя до тех пор, пока значение этого аргумента больше 5. Хотя на первый взгляд может показаться, что такое поведение функции похоже на обычный цикл, на самом деле все работает несколько по-иному: если вы вызовите ее со значением 8, то она выдаст вам 3 сообщения в следующей последовательности: 5, 6, 7. Иначе говоря, функция вызывала саму себя до тех пор, пока значение x было больше 5, и собственно вывод сообщений начала 3-я по уровню получившейся вложенности функция, которая и вывела первое сообщение (в данном случае им стало 5, т.е. уменьшенное на единицу 6).

Чтобы представить себе более наглядно, как работает рекурсивный вызов, дополним эту функцию выводом комментариев, а так же счетчиком глубины рекурсии. Для этого мы, во-первых, задействуем возвращаемое функцией значение, а во-вторых, добавим еще один параметр, который и будет счетчиком. Результат проделанной работы приведен в листинге 6.2.

Листинг 6.2. Рекурсия с комментариями

```
program recurse;
{$APPTYPE CONSOLE}
function recfunc(x, depth: integer) : integer;
begin
 dec(x);
 if x > 5 then begin
   write('Current recursion depth is: ');
   write(depth);
   write(', current x value is: ');
   writeln(x);
   inc(depth);
   depth:=recfunc(x, depth);
 end else writeln('End of recursive calls...');
 write('Current recursion depth is: ');
 write(depth);
 write(', current x value is: ');
```

```
 writeln(x);
 dec(depth);
 result := depth;
end;
begin
 recfunc(8,0);
 readln;
end.
```

Исходный код находится в Demo\Part1\Recurse, там же находится и исполняемый файл recurse.exe, результат работы которого вы можете увидеть на своем экране.

Использование параметров

Параметры в процедурах и функциях могут применяться не только по своему прямому предназначению – для передачи данных подпрограмме, но так же могут быть использованы для возвращения значений. Подобное их использование может быть вызвано, например, необходимостью получить более одного значения на выходе функции. Синтаксис объявления параметров в таком случае несколько отличается от стандартного – перед именем параметра следует использовать ключевое слово var:

```
procedure Circle (square: real; var radius, length: real);
```

Данная процедура принимает «на обработку» одно значение – площадь (square), а возвращает через свои параметры два – радиус (radius) и длину окружности (length). Практическая ее реализация может выглядеть таким образом:

```
procedure Circle (square: real; var radius, length: real);
begin
  radius := sqrt(square / pi); // функция pi возвращает значение числа π
  length := pi * radius * 2;
end;
```

Теперь, чтобы воспользоваться этой функцией, следует объявить в программе 2 переменные, которые будут переданы в качестве аргументов этой процедуре и получат результаты. Их имена не важны, важно лишь, чтобы они были такого же, или совместимого типа, т.е. вещественные, например:

```
var r,l: real;
...
Circle(100,r,l);
```

После вызова функции Circle, переменные r и l получат значения радиуса и длины окружности. Остается их вывести при помощи writeln. Исходный код программы приведен в листинге 6.3.

Листинг 6.3. Процедура с параметрами

```
program params;
{$APPTYPE CONSOLE}
procedure Circle (square: real; var radius, length: real);
begin
  //функция sqrt извлекает корень, а функция pi возвращает значение числа π
  radius := sqrt(square / pi);
  length := pi * radius * 2;
```

```
end;
var
  r,l: real;
begin
  Circle(100,r,l);
  writeln(r);
  writeln(l);
  readln;
end.
```

Запустив такую программу, можно убедиться, что она работает и выводит верные результаты, однако вид у них получается довольно-таки неудобочитаемый, например, длина окружности будет представлена как «3,54490770181103E+0001». Чтобы сделать вывод более удобным для восприятия, нам понадобится функция FloatToStrF. С ее помощью мы можем определить вывод числа на свое усмотрение, например:

```
FloatToStrF(r,ffFixed,12,8)
```

Кроме того, не помешало бы указать, где радиус, а где – длина окружности. Для этого модернизируем строки вывода результатов следующим образом:

```
writeln('Radius is: '+FloatToStrF(r,ffFixed,12,8));
writeln('Length is: '+FloatToStrF(l,ffFixed,12,8));
```

Наконец, не помешало бы сделать программу более полезной, для чего предусмотрим возможность ввода значения площади круга пользователем. В этих целях нам понадобится еще одна переменная (назовем ее s) и выражение для считывания ввода. Не помешает так же приглашение, объясняющее пользователю, что надо делать. В итоге основной блок программы получит следующий вид:

```
...
var
 s,r,l: real;
begin
 write('Input square: ');
 readln(s);
 Circle(s,r,l);
 writeln('Radius is: '+FloatToStrF(r,ffFixed,12,8));
 writeln('Length is: '+FloatToStrF(l,ffFixed,12,8));
 readln;
end.
```

В принципе, это уже лучше, однако не помешало бы добавить обработку возможных ошибок ввода. Скажем, площадь должна быть больше 0. Проверку на то, является ли значение s больше нуля, можно производить непосредственно в основном коде программы, но в целях создания более универсального кода, вынесем ее в подпрограмму. Для этого первой инструкцией процедуры Circle должна быть проверка значения площади:

```
if square <= 0 then exit;
```

Таким образом, в случае, если введенное пользователем значение окажется нулевым или отрицательным, выполнение процедуры будет прекращено. Но возникает другой вопрос: как сообщить программе о том, что вычисления не были выполнены? Пожалуй, в данном случае следовало бы заменить процедуру функцией, которая возвра-

щала бы истину, если вычисления произведены, и ложь в противном случае. Вот что у нас получится:

```
function Circle(square: real; var radius, length: real) : boolean;
begin
 result := false;
 if (square <= 0) then exit;
 radius := sqrt(square / pi);
 length := pi * radius * 2;
 result := true;
end;
```

В начале функции мы определили возвращаемое значение как ложь. В результате, если параметр square не проходит проверку, то функция будет завершена и возвратит именно это значение. Если же проверка будет пройдена, то функция выполнится до конца, т.е. как раз до того момента, когда ее результатом станет истина.

Поскольку программа теперь может получить сведения о том, выполнились ли преобразования на основании возвращаемого функцией Circle булевского значения, остается добавить такую проверку в тело программы. В качестве условия для условного оператора в таком случае подойдет сама функция Circle (на самом деле, условием будет выступать не функция, а как раз возвращаемое ей значение):

```
if Circle(s,r,l) then begin
  // вывод
end else // сообщить об ошибке
```

Результатом проделанной работы будет программа, приведенная в листинге 6.4. Она же находится в Demo\Part1\Params.

Листинг 6.4. Функция с параметрами

```
program params;
{$APPTYPE CONSOLE}
uses sysutils;  //этот модуль соджержит функцию FloatToStrF
function Circle(square: real; var radius, length: real) : boolean;
begin
 result := false;
 if (square <= 0) then exit;
 radius := sqrt(square / pi);
 length := pi * radius * 2;
 result := true;
end;
var
 s,r,l: real;
begin
 write('Input square: ');
 readln(s);
 if Circle(s,r,l) then begin
   writeln('Radius is: '+FloatToStrF(r,ffFixed,12,8));
   writeln('Length is: '+FloatToStrF(l,ffFixed,12,8));
 end else writeln('Data is out of range!');
```

```
 readln;
end.
```

Итак, при помощи ключевого слова var в списке параметров подпрограммы мы можем добиться использования передаваемых аргументов в том блоке, где был произведен вызов данной подпрограммы. В несколько другом аспекте используется ключевое слово const. Фактически, оно объявляет локальную константу, т.е. значение, которое нельзя изменять внутри данной процедуры или функции. Это бывает полезным в том случае, когда такое изменение недопустимо по логике программы и служит гарантией того, что такое значение не будет изменено.

При этом открывается еще одна возможность, связанная с константами, а именно – использование предопределенных значений. Например, можно определить функцию следующим образом:

```
function MyBetterFunc(val1: integer; const val2: integer = 2);
begin
 result := val1*val2;
end;
```

Обращение же к такой функции может иметь 2 варианта: с указанием только одного аргумента (для параметра val1), или же с указанием обоих:

```
x := MyBetterFunc(5); // получим 10
x := MyBetterFunc(5,4); // получим 20
```

Оба вызова будут верными, просто в первом случае для второго параметра будет использовано значение, заданное по умолчанию.

Области видимости

Еще одной важной деталью, касающейся использования подпрограмм, является видимость переменных. Само понятие видимости подразумевает под собой тот факт, что переменная, объявленная в одном месте программы может быть доступна, или наоборот, недоступна, в другом. Прежде всего, это касается подпрограмм: как мы уже успели отметить, переменные, объявленные в заголовке процедур или функций, только в данной процедуре (функции) и будут доступны – на то они и называются локальными:

```
program Project1;
procedure Proc1;
var
 a: integer;
begin
 a := 5; //верно. Локальная переменная a здесь видна
end;
begin
 a := 10; //Ошибка! Объявленная в процедуре Proc1 переменнаая здесь не видна
end.
```

В то же время переменные, объявленные в основном заголовке программы, доступны во всех входящих в нее подпрограммах. Потому они и называются глобальными. Единственное замечание по этому поводу состоит в том, что глобальная переменная должна быть объявлена до функции, т.е. выше ее по коду программы:

```
program Project2;
var
 a: integer; // глобальная переменная a
procedure Proc1;
begin
 a := 5; // верно
 b := 10; // Ошибка! Переменая b на этот момент еще не объявлена
end;
var
 b: integer; // глобальная переменная b
begin
 a := 10; // верно
 b := 5; // тоже верно. Здесь видны все г var
 a: integer; // глобальная переменная
льные переменные
end.
```

Теперь рассмотрим такой вариант, когда у нас имеются 2 переменных с одним и тем же именем. Разумеется, компилятор еще на стадии проверки синтаксиса не допустит, чтобы в программе были объявлены одноименные переменные в рамках одного диапазона видимости (скажем, 2 глобальных переменных X, или 2 локальных переменных X в одной и той же подпрограмме). Речь в данном случае идет о том, что произойдет, если в одной и той же программе будет 2 переменных X, одна – глобальная, а другая – локальная (в какой-либо подпрограмме). Если с основным блоком программы все ясно – в нем будет присутствовать только глобальная X, то как быть с подпрограммой? В таком случае в действие вступает правило близости, т.е. какая переменная ближе (по структуре) к данному модулю, та и есть верная. Применительно к подпрограмме ближней оказывается локальная переменная X, и именно она будет задействована внутри подпрограммы.

```
program Project3;
var
 X: integer;
procedure Proc1;
var
 X: integer;
begin
 X := 5; // Здесь значение будет присвоено локальной переменной X
end;
begin
 X := 10; // Здесь же значение будет присвоено голобальной переменной X
end.
```

Таким образом, мы внесли ясность в вопрос видимости переменных. Что касается видимости подпрограмм, то она определяется аналогичным образом: подпрограммы, объявленные в самой программе, видны всюду. Те же подпрограммы, которые объявлены внутри процедуры или функции, доступны только внутри нее:

```
program Project1;
procedure Proc1;
 procedure SubProc;
```

```
 begin
 end;
begin
 SubProc; // Верно. Вложенная процедура здесь видна.
end;
begin
 Proc1; // Верно. Процедура Proc1 объявлена в зоне глобальной видимости
 SubProc; // Ошибка! Процедура SubProc недоступна за пределами Proc1.
end.
```

Наконец в том случае, когда имена встроенной и некой глобальной процедуры совпадают, то, по аналогии с переменными, в области видимости встроенной процедуры, именно она и будет выполнена.

Видимость в модулях

Все то, что мы уже рассмотрели, касалось программ, умещающихся в одном единственном файле. На практике же, особенно к тому моменту, когда мы перейдем к визуальному программированию, программы будут включать в себя множество файлов. В любом случае, программа на Object Pascal будет иметь уже изученный нами файл проекта – dpr, или основной модуль программы. Все прочие файлы будут располагаться в других файлах, или модулях (units), с типичным для Pascal расширением pas. При объединении модулей в единую программу возникает вопрос видимости переменных, а так же процедур и функций в различных модулях.

Для начала вернемся к рассмотрению структуры модуля, которая имеет ряд отличий от структуры программы. Итак, в простейшем случае, модуль состоит из названия, определяемого при помощи ключевого слова unit, и 2 секций – interface и implementation. Так вот как раз первая секция, interface, и служит для определения (декларации) типов данных, переменных, функций и процедур данного модуля, которые должны быть доступны за пределами данного модуля.

Чтобы лучше в этом разобраться, создадим программу, состоящую из 2 модулей – основного (dpr) и дополнительного (pas). Для этого сначала создайте новый проект типа Console Application, а затем добавьте к нему модуль, для чего из подменю File → New выберите пункт Unit. После этого сохраните проект, щелкнув по кнопке Save All (или File → Save All). Обратите внимание, что первым будет предложено сохранить не файл проекта, а как раз файл дополнительного модуля. Назовем его extunit.pas, а сам проект – miltiunits (см. Demo\Part1\Visibility). При этом вы увидите, что в части uses файла проекта произошло изменение: кроме постоянно добавляемого модуля SysUtils, появился еще один модуль – extunit, т.е. код стал таким:

```
uses
  SysUtils,
  extunit in 'extunit.pas';
```

Мы видим, что Delphi автоматически добавила пояснение, в каком файле находится подключаемый модуль. Это вызвано тем, что если о расположении собственных модулей Delphi все известно, то пользовательские модули могут находиться где угодно на жестком диске ПК. Но в данном случае мы сохранили и файл программы, и подключаемый модуль в одном каталоге, следовательно, их пути совпадают, и данное указание можно было бы опустить:

```
uses
  SysUtils, extunit;
```

Тем не менее, оставим код как есть, и приступим к разработке модуля extunit. В нем, в части implementation, напишем 2 процедуры – ExtProc1 и ExtProc2. Обе они будут делать одно и то же – выводить строку со своим названием. Например, для первой:

```
writeln('ExtProc1');
```

Теперь вернемся к главному модулю программы и попробуем обратиться к процедуре ExtProc1:

```
...
begin
 ExtProc1;
end.
```

Попытка компиляции или запуска такой программы приведет к ошибке компилятора «Undeclared identifier», что означает «неизвестный идентификатор». И действительно, одного лишь описания процедуры недостаточно, чтобы она была доступна вне своего модуля. Так что перейдем к редактированию extunit и в секции interface напишем строку:

```
procedure ExtProc1;
```

Такая строка, помещенная в секцию interface, является объявлением процедуры ExtProc1, и делает ее видимой вне данного модуля. Отметим, что в секции interface допускается лишь объявлять процедуры, но не определять их (т.е. тело процедуры здесь будет неуместно). Еще одним полезным эффектом от объявления процедур является то, что таким образом можно обойти такое ограничение, как необходимость определения подпрограммы до ее вызова. Иначе говоря, поскольку в нашем файле уже есть 2 процедуры, ExtProc1и ExtProc2, причем они описаны именно в таком порядке – сначала ExtProc, а потом ExtProc2, то выведя объявление ExtProc2 в interface, мы сможем обращаться к ExtProc2 из ExtProc1, как это показано в листинге 6.5:

Листинг 6.5. Объявление процедур в модуле

```
unit extunit;
interface
procedure ExtProc1;
procedure ExtProc2;
implementation
procedure ExtProc1;
begin
 writeln('ExtProc1');
 ExtProc2; // Если объявления не будет, то компилятор выдаст ошибку
end;
procedure ExtProc2;
begin
 writeln('ExtProc2');
end;
end.
```

Отметим, что теперь процедуры ExtProc2, так же, как и ExtProc1, будет видна не только по всему модулю extunit, но и во всех использующей этот модуль программе multiunits.

Разумеется, все, что было сказано о процедурах, верно и для функций. Кроме того, константы и переменные, объявленные в секции interface, так же будут видны как во всем теле модуля, так и вне него. Остается лишь рассмотреть вопрос пересечения имен, т.е. когда имя переменной (константы, процедуры, функции) в текущем модуле совпадает с таковым в подключенном модуле. В этом случае вновь вступает в силу правило «кто ближе, тот и прав», т.е. будет использоваться переменная из данного модуля. Например, если в extunit мы объявим типизированную константу Z, равную 100, а в multiunits – одноименную константу, равную 200, то обратившись к Z из модуля extunit, мы получим значение 100, а из multiunits – 200.

Если же нам в multiunits непременно понадобится именно та Z, которая находится в модуле extunit, то мы все-таки можем к ней обратиться, для чего нам пригодится точечная нотация. При этом в качестве имени объекта указывают название модуля:

```
extunit.Z;
```

Именно таким образом можно явно ссылаться на переменные, функции и процедуры, находящиеся в других модулях.

Некоторые стандартные функции

В Object Pascal, как уже отмечалось, имеются огромное количество стандартных процедур и функций, являющихся составной частью языка, и с некоторыми мы уже знакомы (например, приведенные в табл. 5.1 и 5.2 функции преобразования). Детальное описание всех имеющихся в Object Pascal процедур и функций можно получить в справочной системе Delphi, однако мы все-таки рассмотрим здесь некоторые из них, чтобы составить общее представление – см. таблицу 6.1.

Таблица 6.1. Некоторые стандартные процедуры и функции Delphi

Синтаксис	Группа	Модуль	Описание
function Abs(X);	арифметические	System	Возвращает абсолютное значение числа
procedure ChDir(const S: string);	управления файлами	System	Изменяет текущий каталог
function Concat(s1 [, s2,..., sn]: string): string;	строковые	System	Объединяет 2 и более строк в 1
function Copy(S; Index, Count: Integer): string;	строковые	System	Возвращает часть строки
function Cos(X: Extended): Extended;	тригонометрические	System	Вычисляет косинус угла
procedure Delete(var S: string; Index, Count: Integer);	строковые	System	Удаляет часть строки
function Eof(var F): Boolean;	ввод-вывод	System	Проверяет, достигнут ли конец файла
procedure Halt [(Exitcode: Integer)];	управления	System	Инициирует досрочное прекращение программы
function High(X);	диапазона	System	Возвращает максимальное зна-

			чение из диапазона
procedure Insert(Source: string; var S: string; Index: Integer);	строковые	System	Вставляет одну строку в другую
function Length(S): Integer;	строковые	System	Возвращает длину строки или количество элементов массива
function Ln(X: Real): Real;	арифметические	System	Возвращает натуральный логарифм числа (Ln(e) = 1)
function Low(X);	диапазона	System	Возвращает минимальное значение из диапазона
procedure New(var P: Pointer);	размещения памяти	System	Создает новую динамическую переменную и назначает указатель для нее
function ParamCount: Integer;	командной строки	System	Возвращает количество параметров командной строки
function ParamStr(Index: Integer): string;	командной строки	System	Возвращает указанный параметр из командной строки
function Pos(Substr: string; S: string): Integer;	строковые	System	Ищет вхождение указанной подстроки в строку и возвращает порядковый номер первого совпавшего символа
procedure RmDir(const S: string);	ввод-вывод	System	Удаляет указанный подкаталог (должен быть пустым)
function Slice(var A: array; Count: Integer): array;	разные	System	Возвращает часть массива
function UpCase(Ch: Char): Char;	символьные	System	Преобразует символ в верхний регистр
function LowerCase(const S: string): string;	строковые	SysUtils	Преобразует ASCII-строку в нижний регистр
procedure Beep;	разные	SysUtils	Инициирует системный сигнал
function CreateDir(const Dir: string): Boolean;	управления файлами	SysUtils	Создает новый подкаталог
function CurrentYear: Word;	даты и времени	SysUtils	Возвращает текущий год
function DeleteFile(const FileName: string): Boolean;	управления файлами	SysUtils	Удаляет файл с диска
function ExtractFileExt(const FileName: string): string;	имен файлов	SysUtils	Возвращает расширение файла
function FileExists(const FileName: string): Boolean;	управления файлами	SysUtils	Проверяет файл на наличие
function IntToHex(Value: Integer; Digits: Integer): string;	форматирования чисел	SysUtils	Возвращает целое в шестнадцатеричном представлении
function StrPCopy(Dest: PChar; const Source: string): PChar;	строковые	SysUtils	Копирует Pascal-строку в C-строку (PChar)
function Trim(const S: string): string;	строковые	SysUtils	Удаляет начальные и конечные пробелы в строке
function TryStrToInt(const S:	преобразо-	SysUtils	Преобразует строку в целое

string; out Value: Integer): Boolean;	вания типов		
function ArcCos(const X: Extended): Extended;	тригонометрические	Math	Вычисляет арккосинус угла
function Log2(const X: Extended): Extended;	арифметические	Math	Возвращает логарифм по основанию 2
function Max(A,B: Integer): Integer;	арифметические	Math	Возвращает большее из 2 чисел
function Min(A,B: Integer): Integer;	арифметические	Math	Возвращает меньшее из 2 чисел

Те функции, которые имеются в модуле System, являются основными функциями языка, и для их использования не требуется подключать к программе какие-либо модули. Все остальные функции и процедуры можно назвать вспомогательными, и для их использования следует подключить тот или иной модуль, указав его в uses, например, как это делает Delphi уже при создании новой программы с SysUtils:

```
uses: SysUtils;
```

Что касается практического применения той или иной функции, то оно определяется, прежде всего, той группой, к которой данная функция относится. Например, арифметические функции используются для различных математических расчетов, строковые используются для манипуляций со строками и т.д. Разумеется, в каждой категории имеется множество других функций, помимо тех, что приведены в таблице 6.1, однако по ней можно получить общее представление о том, что есть в распоряжении Delphi-программиста.

Функции в действии

В целом мы уже ознакомились с несколькими десятками предопределенных процедур и функций, а так же умеем создавать собственные. Пора применить полученные знания на практике, для чего вновь вернемся к программе, рассмотренной в главе, посвященной операторам – игре «Угадай-ка». В ней, по сути, был реализован только один из рассмотренных в самом начале книги алгоритмов – угадывания числа. Что касается алгоритма управления, то на тот момент мы оставили его без внимания.

Но прежде, чем вносить в программу изменения, определимся с тем, что мы все-таки хотим получить в итоге. Допустим, что мы хотим сделать следующие вещи:

1. Реализовать-таки возможность повторного прохождения игры без перезапуска программы;
2. Добавить немного «геймплея». Иначе говоря, введем уровни сложности и подсчет очков. Новые уровни можно реализовать как повторное прохождение игры с увеличением сложности (скажем, за счет расширения диапазона загадываемых значений);
3. В продолжение п. 2 добавить еще и таблицу рекордов, которая будет сохраняться на диске.

Поскольку часть работы уже выполнена, то для того, чтобы приступить к разработке новой версии игры (назовем ее «Угадай-ка 2.0»), мы не будем как обычно создавать новый консольный проект в Delphi, а откроем уже существующий (Ugadaika) и сохраним его под новым именем, скажем, Ugadaika2, и в новом каталоге. Таким обра-

зом, мы уже имеем часть исходного кода, отвечающую за угадывание, в частности, цикл while (см. листинг 4.5). Этот фрагмент логичнее всего выделить в отдельную процедуру, вернее даже функцию, которая будет возвращать число попыток, сделанное пользователем. Для этого создадим функцию, которая будет принимать в качестве аргумента число, которое следует угадать, а возвращаемым значением будет целое, соответствующее числу попыток. Ее объявление будет таким:

```
function GetAttempts(a: integer):integer;
```

Данная функция так же должна иметь в своем распоряжении переменную, необходимую для ввода пользователем своего варианта ответа. Еще одна переменная нужна для подсчета результата, т.е. количества попыток. В качестве первой можно было бы использовать глобальную переменную (b), однако во избежание накладок, для локального использования в функции следует использовать локальную же переменную. Что касается переменной-счетчика, то для нее как нельзя лучше подходит автоматическая переменная result. Еще одним изменением будет использование цикла repeat вместо while. Это вызвано тем, что с одной стороны, тем, что хотя бы 1 раз пользователь должен ввести число, т.е. условие можно проверять в конце цикла, а с другой мы можем избавиться от присвоения лишнего действия, а именно – присвоения заведомо ложного значения переменной b. Ну и еще одно дополнение – это второе условие выхода, а именно – ограничение на число попыток, которое мы установим при помощи константы MAXATTEMPTS:

```
const MAXATTEMPTS = 10;
```

В результате код функции получится таким, как представлено в листинге 6.6.

Листинг 6.6. Функция GetAttempts

```
function GetAttempts(a: integer):integer;
var
 b: integer;
begin
 Result:=0;
 repeat
   inc(Result); // увеличиваем счетчик числа попыток
   write(#13+#10+'?:');
   read(b);
   if (b>a) then begin
     write('Too much!');
     continue;
   end;
   if (b<a) then begin
     write('Too little!');
     continue;
   end;
 until (a=b) or (result=10); // цикл repeat выполняется пока условие ложно
end;
```

Теперь, когда подготовительная работа сделана, можно браться за реализацию намеченных изменений. Прежде всего, в теле программы нам потребуется цикл, который как раз и будет обеспечивать логику исполнения программы. Для него нам так же понадобятся переменные. В частности, нужны счетчик цикла, устанавливающий те-

кущий уровень сложности, так же нужны переменные для хранения набранных очков и числа попыток, и, кроме того, не помешает заранее определить файловую переменную для таблицы рекордов и строковую – для ввода имени «рекордсмена». Итого мы получаем следующий список переменных перед основным блоком программы:

```
var
 level, score, attempt: integer;
 f: TextFile;
 s: string;
```

Теперь инициализируем счетчик псевдослучайных чисел (т.е. оставим randomize на месте) и инициализируем нулем значения счета и уровня:

```
score:=0;
level:=1;
```

Наконец, напишем цикл для основного блока программы. Этот цикл должен быть выполнен хотя бы один раз и будет продолжать выполняться до тех пор, пока число попыток в последнем уровне было меньше максимально допустимого. В результате получаем цикл repeat со следующим условием:

```
until attempt>=MAXATTEMPTS;
```

В самом цикле нам потребуется, прежде всего, выводить информацию о текущем уровне, а так же о диапазоне отгадываемых чисел. После этого надо будет получить число попыток при помощи функции GetAttempts, вычислить набранные очки и сообщить о них пользователю, после чего увеличить счетчик цикла на 1 и перейти к следующей его итерации. В результате мы получим следующий фрагмент кода:

```
repeat
  writeln('Level '+IntToStr(level)+':');
  writeln('From 0 to '+IntToStr(level*100));
  attempt:=GetAttempts(random(level*100+1));
  score:=score+(MAXATTEMPTS-attempt)*level;
  writeln(#10+'You current score is: '+IntToStr(score));
  inc(level);
until attempt>MAXATTEMPTS;
```

После завершения работы цикла, т.е. когда пользователь хоть раз истратит на отгадывание все 10 попыток, следует сообщить итоговый результат и сравнит его с предыдущим значением, которое следует считать из файла. Файл мы назовем records.txt, и сопоставим с переменной f:

```
AssignFile(f,'record.txt');
```

Но прежде, чем попытаться что-либо прочитать из этого файла, необходимо убедиться, что такой файл уже есть, а если нет – то создать его, записав в него некий минимальный результат.

```
if not FileExists('record.txt') then begin
  Rewrite(f);
  writeln(f,'0'); // первая строка содержит число-рекорд
  writeln(f,'None'); // а вторая - имя последнего победителя
  CloseFile(f);
end;
```

Теперь можно считать этот файл. Правда, мы упустили из виду, что нам здесь тоже нужна переменная – для считывания предыдущего рекорда. В то же время, на данный момент мы уже имеем 2 ненужных для дальнейшей работы программы переменных – attempt и level, так что вполне можно воспользоваться любой из них для этих целей. Таким образом, мы получим следующий код:

```
Reset(f);
readln(f, attempt);
readln(f,s);
writeln(#10+'BEST SCORE: '+IntToStr(attempt)+' by '+s);
CloseFile(f);
```

Ну и последнее, чего нам остается – это проверить, является ли новое значение выше рекорда, и если да – то записать новый рекорд в файл, не забыв спросить имя игрока:

```
if attempt<score then begin
  writeln('NEW BEST SCORE!!!');
  writeln('You name?:');
  readln;
  readln(s);
  Rewrite(f);
  writeln(f,score);
  writeln(f,s);
  CloseFile(f);
end;
```

Вот, собственно, и все. Полный код получившейся программы можно увидеть на листинге 6.7, или же в файле проекта в каталоге Demo\Part1\Ugadaika2.

Листинг 6.7. Программа угадай-ка, окончательный вариант

```
program ugadaika2;
{$APPTYPE CONSOLE}
uses
  SysUtils;
const MAXATTEMPTS = 10;
function GetAttempts(a: integer):integer;
var
 b: integer;
begin
 Result:=0;
 repeat
   inc(Result);
   write(#13+#10+'?:');
   read(b);
   if (b>a) then begin
     write('Too much!');
   end;
   if (b<a) then begin
     write('Too little!');
   end;
```

```
 until (a=b) or (result=MAXATTEMPTS);
end;
var
 level, score, attempt: integer;
 f: TextFile;
 s: string;
begin
  randomize;
  score:=0;
  level:=1;
  repeat
    writeln('Level '+IntToStr(level)+':');
    writeln('From 0 to '+IntToStr(level*100));
    attempt:=GetAttempts(random(level*100+1));
    score:=score+(MAXATTEMPTS-attempt)*level;
    writeln(#10+'You current score is: '+IntToStr(score));
    inc(level);
  until attempt>=MAXATTEMPTS;
  write(#10+'TOTAL SCORE: '+IntToStr(score+level-1));
  AssignFile(f,'record.txt');
  if not FileExists('record.txt') then begin
    Rewrite(f);
    writeln(f,'0');
    writeln(f,'None');
    CloseFile(f);
  end;
  Reset(f);
  readln(f,attempt);
  readln(f,s);
  writeln(#10+'BEST SCORE: '+IntToStr(attempt)+' by '+s);
  CloseFile(f);
  if attempt<score then begin
    writeln('NEW BEST SCORE!!!');
    writeln('You name?:');
    readln;
    readln(s);
    Rewrite(f);
    writeln(f,score);
    writeln(f,s);
    CloseFile(f);
  end;
  sleep(3000);
end.
```

В завершение отметим, что эта программа использует использование не только функций, но и констант, глобальных и локальных переменных, а так же циклов и операций файлового ввода-вывода. Таким образом, на текущий момент мы познако-

мились со всеми основами обычного, процедурного программирования. Пора двигаться дальше – к объектно-ориентированному программированию в Object Pascal!

Часть II. Объектно-ориентированное программирование

Введение в ООП

Современные программы, как правило, сложны и велики по объему исходного кода, и использования лишь обычных процедурных методов, описанных в первой части, недостаточно для написания полноценных Windows-приложений. Поэтому здесь мы рассмотрим объектно-ориентированное программирование, а так же остановимся на вопросах планирования программ и цикла их разработки.

Суть современного программирования

Если в 80-х годах повсеместно применялось, в основном, процедурное программирование, то с приходом в начале 90-х графических сред наподобие Windows, сопровождающееся значительным усложнением структуры и размеров программного кода, возник вопрос использования новых, более прогрессивных средств. Таковыми оказались объектно-ориентированные языки программирования – такие, как C++, а немного позже к ним добавились новые, например, язык Java или полностью переработанные старые – Object Pascal.

ПРИМЕЧАНИЕ

> Собственно говоря, язык C++ так же является переработанным вариантом процедурного языка C, только появился он гораздо раньше – в 1982 году. Что касается языка Pascal, то в принципе, первой попыткой переложения его на объектно-ориентированное применение можно считать такую ветвь, как язык Ada.

Увеличение сложности программ привело не только к появлению новых инструментов, упрощающих разработку, но и повлекло за собой кардинальной пересмотр путей подхода к решению поставленной задачи. Так, если во времена DOS, программист, получив задание, сразу приступал к кодированию, то теперь этому предшествует этап анализа и предварительной разработки. Более того, теперь существует несколько специализаций, причем не всех их можно назвать программистами в привычном смысле этого слова. Это системные аналитики, дизайнеры интерфейса, программные инженеры, собственно «кодеры» и другие. Впрочем, такое многообразие больше характерно для крупных коллективов, решающих объемные задачи. В более простом и прагматичном случае все эти обязанности выполняет один человек.

Итак, что же делать разработчику, получившему задачу по созданию того или иного приложения? Прежде всего, требуется проанализировать поставленную задачу, т.е. определить, какой вариант решения поставленного вопроса будет наиболее правильным, и что понадобится для его реализации. В простейшем случае понадобятся только компьютер и установленная на нем среда Delphi. Для решения более сложных задач могут понадобиться другие программы (начиная от приложений баз данных и заканчивая графическим редактором), а возможно – и другие специалисты, как-то

другие программисты, 3D-моделисты, художники, технические писатели и т.д. На этом этапе так же важно взаимодействие с заказчиком проекта для внесения полной ясности в понимание того, чего все-таки требуется от программы, возможности внесения изменений, увеличения или уменьшения функциональности и т.п.

После того, как ситуация прояснится в достаточной степени для того, чтобы было понятно что и зачем надо делать, можно будет окончательно определится с тем, что понадобится для работы и составить общую схему программы. После этого можно перейти к детальной разработке программы.

На этапе детальной разработки следует перейти к определению конкретных составляющих программы – от общей структуры проекта до отдельных диалоговых окон, схемы поведения, вариантов вывода отчетов и т.д. Следует так же продумать реализацию тех или иных алгоритмов, структуру хранения данных и прочие подобные вопросы. Детальная проработка этих вопросов до того, как начнется собственно процесс написания программы, существенно ускорит весь процесс и, возможно, избавит от многократного переписывания целых фрагментов программы.

Наконец, когда ясны все цели поставленной задачи и намечены пути их решения, а так же есть ясность в том, что и как должно выглядеть и работать, можно приступать к написанию исходного кода – кодированию. И хотя это далеко не последний этап создания программы, процесс написания исходного кода, однажды начавшись, будет продолжаться во всех последующих этапах. А следующим этапом будет тестирование программы на работоспособность, выявление различных ошибок, доводке отдельных модулей и т.д. Причем чем больше программа по объему, тем дольше и сложнее ее тестировать, причем зависимость здесь отнюдь не линейная: увеличение размера программы вдвое увеличивает число потенциальных ошибок, как минимум, вчетверо.

Последним, и самым продолжительным этапом разработки, является сопровождение программы. Дело в том, что никакое тестирование не выловит все огрехи кода в большой программе. Поэтому выявление пользователями различных «недокументированных возможностей», а, попросту говоря – ошибок, может продолжаться ровно до тех пор, пока данное творение используется вообще. Кроме того, заказчику со временем наверняка захочется получить какие-либо дополнительные, ранее не предусмотренные возможности, изменить вид или тип поведения для тех или иных модулей и т.д.

Разумеется, для всего этого придется вновь и вновь возвращаться к написанию исходного кода. Таким образом, можно сказать, что отладка и сопровождение – поздние фазы этапа кодирования. В то же время, не будем забывать, что в целом программа готова уже после того, как закончился этап основного кодирования – именно там, где мы его определили, т.е. до начала тестирования. Но, опять-таки, тестирование тоже бывает разным – в практике выделяют несколько его фаз, а именно:

- Альфа-тестирование. Это предварительный вариант программы, на этапе выпуска альфа-версии функциональность продукта, как правило, не является полной;
- Бета-тестирование. После того, как с одной стороны, исправлены ошибки, выявленные при альфа-тестировании, а с другой – завершено функциональное наполнение продукта, выпускают бета-версию;

- Финишная подгонка, или Pre-Release. С некоторых пор вошло в моду предварять выход окончательной версии предварительным выпуском (PR, RC). На этот момент программа должна на 100% соответствовать тому, что ожидается в окончательной версии (Release), суть же выпуска такой версии – убедиться еще раз, что все в порядке.

Фактически, можно констатировать, что этап основного кодирования завершается одновременно с выпуском бета-версии, т.е. когда программа уже умеет делать все, что ей положено, возможно, за исключением тех случаев, когда она не может что-то сделать из-за не выявленной ранее ошибки.

Использование объектно-ориентированного подхода как раз и служит для того, чтобы разработчику было проще ориентироваться в исходном коде, и минимизировать, или даже локализовать допущенные при кодировании ошибки.

Классы и объекты

Основой объектно-ориентированного программирования являются сами объекты. Каждый объект обладает характеристиками (свойствами, атрибутами) и моделью поведения (методами, реакцией не события). К примеру, объект, описывающий машину, может иметь свойство, отвечающее за то, что с ней происходит. Допустим, машина может ехать или стоять на месте, соответственно для определения того, стоит машина на месте или движется, может отвечать некое свойство булевского типа.

В ООП описания объектов называются классами (class). Фактически, класс является дальнейшим развитием уже знакомого нам типа данных – записи. Но если запись имела только свойства (поля) и ничего более, то классы могут содержать все, что требуется для полноценного объекта. При этом класс может быть сколь угодно большим и иметь в качестве своих свойств другие классы. Например, класс «машина» может иметь в своем описании такие свойства, как двигатель, шасси и им подобные, представленные другими классами. При этом для каждого такого класса так же могут быть определены собственные свойства и методы.

Самими же объектами в программе называют не классы, а их экземпляры. Т.е. если мы имеем класс TCar, то его экземпляр, представленный в виде переменной типа TCar (скажем, MyCar), как раз и будет объектом. Чтобы использовать объект, его надо предварительно создать. Делается это при помощи специального метода, который определен для всех классов – Create, называемый так же конструктором класса. Соответственно, чтобы получить объект, являющийся экземпляром класса, мало объявить переменную – надо еще и проинициализировать его при помощи конструктора:

```
var MyCar: TCar;
...
MyCar.Create;
```

Разумеется, каждый класс может иметь множество своих экземпляров. Скажем, кроме объекта MyCar, могут быть еще BossCar, AnyCar, Truck23, Bus144 и т.д. Все эти переменные называются переменными-экземплярами класса.

Другой составляющей ООП являются 3 концепции, применяемые к классам – инкапсуляция, наследование и полиморфизм. И хотя в предыдущей части книги мы уже рассмотрели все эти термины, вспомним их суть здесь еще раз, причем уже применительно к практическому применению классов.

Так, инкапсуляция, или объединение в единое целое всего того, что относится к определенному объекту, на практике выливается в возможность объединить в одном объекте свойства и методы. Наследование позволяет, единожды создав некий обобщенный класс, пользоваться его свойствами и методами из других классов, созданных на его основе – в классах-потомках. Ну а применение полиморфизма на практике позволит переопределить свойства или изменить поведение отдельных методов родительского класса в этих самых классах-потомках.

В результате всего этого достигается более компактный и понятный код. В частности, благодаря инкапсуляции, для изменения состояния объекта достаточно обратиться к его же методу, а не использовать отдельную функцию. Наследование – это замечательная вещь для многократного использования единожды написанного кода. Ну а полиморфизм позволяет в родственных классах одинаково называть методы, отличающиеся в программной реализации, но одинаковые по своей сути.

Структура класса и видимость

Для объявления класса используют ключевое слово class. При этом, как и любые другие пользовательские типы, объявлению класса или классов всегда предшествует другое ключевое слово – type. Далее следует перечисление всех его полей, свойств и методов, сгруппированных по 4 разделам, определяющих уровень видимости. В результате для описания класса мы имеем следующую структуру:

```
type <Имя класса> = class (<Имя родительского класса>)
 private
  <частные описания>
 protected
  <защищенные описания>
 public
  <общедоступные описания>
 published
  <опубликованные описания>
end;
```

В представленной структуре описаниями являются объявления свойств и методов, при этом большинство составных частей в объявлении класса являются необязательными. В частности, если класс не имеет специфического предка, то имя родительского класса можно опустить, при этом подразумевается, что такой класс происходит от общего для всех классов предка – TObject. Что касается секций, определяющих уровень видимости, то их в ряде случаев так же можно опустить. При этом будет подразумеваться, что все объявленные в классе свойства и методы будут принадлежать к категории опубликованных (published). Таким образом, следующие объявления классов TClass1 и TClass2 будут полностью эквивалентны:

```
type TClass1 = class (TObject)
 published
  Property1: integer;
end;
type TClass2 = class
  Property1: integer;
end;
```

Что касается области видимости, называемых так же правами доступа то, как следует из синтаксиса определения класса, их может быть 4 вида. Так, разделы private и protected содержат защищенные описания, а public и published – общедоступные. Более тонкие различия между этими категориями заключаются в следующем:

- Категория Private является наиболее защищенной. Доступ к помещенным в эту группу свойствам и методам можно получить только в том же самом модуле;
- В категорию Protected помещают те свойства класса, доступ к которым возможен из дочерних классов, объявленных так же и вне данного модуля;
- Раздел Public содержит свойства и методы, которые должны быть видны во всех местах программы, где используется данный класс, т.е. «публичные».

Что касается опубликованных (published) свойств, то эта категория специально создана для библиотеки VCL. Эти те самые свойства, которые отображаются в инспекторе объекта, в остальном они аналогичны публичным свойствам. Все свойства, категория которых в классе явно не задана, считаются опубликованными.

Теперь обратимся собственно к содержимому этих групп. В их качестве могут выступать поля – по аналогии с записями, а так же методы – подпрограммы (как функции, так и процедуры). При этом если имена классов принято начинать с буквы T, то имена полей обычно начинают с буквы F (field). Что касается названий методов, то для их наименования руководствуются теми же принципами, что и при написании обычных подпрограмм, т.е. дают название, характеризующее выполняемое данным методом действие. Для примера создаем класс TUser, содержащий имя учетной записи (login) и пароль (password), а так же некий метод, проверяющий имя и пароль:

```
type TUser = class
  fLogin: string;
  fPassword: string;
  function Connect: boolean;
end;
```

При реализации метода – в данном случае функции Connect, следует указывать имя класса, к которому она принадлежит. Например, в нашем случае она может выглядеть следующим образом:

```
function TUser.Connect: boolean;
var
 s,p: string;
begin
 writeln('Input username:');
 readln(s);
 writeln('Input password:');
 readln(p);
 result:= (s=fLogin) and (p=fPassword);
end;
```

Фактически, составление подпрограммы как метода класса, ничем не отличается от написания обычной процедуры или функции, с той лишь разницей, что перед ее на-

званием указывается имя класса, а в самой функции в качестве переменных доступны все поля этого класса – в данном случае fLogin и fPassword.

Кроме полей и методов, существуют собственно свойства (properties), являющиеся чем-то средним: они могут использовать, скажем, поле, для хранения значения, и функцию для его изменения. Фактически, свойства реализуют механизм доступа к полям. В таком случае поля, как правила, делают защищенными (private), а свойства – общедоступными (published). Допустим, в нашем классе TUser, который хранит некую информацию о пользователе, можно реализовать процедуру смены пароля таким образом, чтобы при попытке установки нового пароля он проверялся на допустимость – скажем, подходил по количеству знаков. В таком случае можно создать свойство Password, которое будет обращаться непосредственно к полю fPassword в случае, когда пароль надо просто считать, и вызывать процедуру проверки при попытке изменения этого поля. Попутно создадим свойство Login, но без процедуры проверки. Объявление такого класса может выглядеть следующим образом:

```
type TUser = class
  private
    fLogin: string;
    fPassword: string;
    procedure setPassword(newpass: string);
  published
    property Login: read fLogin write fLogin;
    property Password: read fPassword write setPassword;
    function Connect: boolean;
end;
```

Обратите внимание, что поля теперь вынесены в защищенную часть класса, там же определена и процедура проверки пароля. Что касается свойств и метода Connect, то они находятся в общедоступной секции.

При обращении к свойству Password на чтение (read) будет автоматически возвращаться значение, хранящееся в поле fPassword, а при попытке записать (write) в поле новое значение будет вызываться процедура setPassword. Сама эта процедура может выглядеть следующим образом:

```
procedure TUser.setPassword(newpass: string);
begin
 if Length(newpass)>3 then fPassword:=newpass
   else writeln('Error! Password is too short!');
end;
```

Эта подпрограмма проверяет новое значение пароля на длину, и если оно содержит более 3 символов, то присваивается полю fPassword, в противном случае значение поля не меняется, и выводится сообщение об ошибке. В самой же программе обращение к свойству класса выглядит точно так же, как к полю записи, т.е. с использованием доступа через точку:

```
User.Login:='Serge';
```

Пример программы, использующей класс TUser, включая описание самого класса вместе со всеми его полями, свойствами и методами, представлен в листинге 7.1 (см. так же Demo\Part2\ClassProp).

Листинг 7.1. Класс со свойствами

```
program classprop;
{$APPTYPE CONSOLE}
type TUser = class
  private
    fLogin: string;
    fPassword: string;
    procedure setPassword(newpass: string);
  published
    property Login: string read fLogin write fLogin;
    property Password: string read fPassword write setPassword;
    function Connect: boolean;
end;
procedure TUser.setPassword(newpass: string);
begin
 if Length(newpass)>3 then fPassword:=newpass
   else writeln('Error! Password is too short!');
end;
function TUser.Connect:boolean;
var
 s,p: string;
begin
 writeln('Input username:');
 readln(s);
 writeln('Input password:');
 readln(p);
 result:= (s=fLogin) and (p=fPassword);
end;
var
 User: TUser;
 s,p: string;
begin
 User:=TUser.Create;
 writeln('Create username');
 readln(s);
 User.Login:=s;
 writeln('Create password');
 readln(p);
 User.Password:=p;
 writeln('Username is: '+User.Login);
 writeln('Password is: '+User.Password);
 writeln('Try to login...');
 while User.Password<>'' do begin
   if User.Connect then begin
     writeln('All Ok.');
     break;
   end else writeln('Incorrect data. Try again!');
```

```
 end;
 User.Destroy;
 readln;
end.
```

В начале здесь создается объект User, являющийся экземпляром класса TUser, после чего его полям fLogin и fPassword, посредством соответствующих свойств назначаются значения. При этом неявно задействуются методы write этих свойств. После чего данные считываются, опять-таки через свойства Login и Password, на сей раз будут использованы методы read. После этого, если пароль был задан, начинается цикл, для выхода из которого пользователю надо ввести верные имя и пароль, для чего использован метод Connect.

Обратите в этом листинге внимание на то, что в подпрограммах, являющихся членами класса, напрямую используются поля, а вне класса задействуются свойства. Хотя в данном случае это и не обязательно, поскольку все находится в одном модуле, а защищенные поля видны в нем повсеместно, в основном теле программы обращения идут только к опубликованным свойствам и методам, что являются более типичным вариантом для реальных приложений.

Конструкторы и деструкторы

В самом начале предыдущего примера мы использовали метод Create для создания экземпляра класса TUser, а в конце – метод Destroy для его удаления, хотя сами эти методы определены в классе не были. Но вспомним, что в ООП существует такое понятие, как наследование, а все классы в Delphi происходят от класса TObject. Именно по наследству от TObject классу TUser эти методы и достались.

Метод Create является конструктором класса, т.е. подпрограммой, создающий объект (экземпляр класса) в памяти. В том случае, если при создании экземпляра класса никаких дополнительных действий производить не требуется, используют метод Create, доставшийся от родительского класса. Если же при создании объекта требуется выполнить какие-либо действия, то метод Create переписывают заново. Например, если при инициализации класса TUser нам надо было бы задать какие-либо имя пользователя и пароль по умолчанию, то мы могли бы определить конструктор заново:

```
type TUser = class
  ...
  constructor Create;
end;
constructor TUser.Create;
begin
 inherited Create;
 fLogin:='User';
 fPassword:='password';
end;
```

Обратите внимание на то, что, во-первых, конструктор – это не процедура и не функция, а именно конструктор, и задается, соответственно, при помощи ключевого слова constructor, а во-вторых, в самом теле подпрограммы-конструктора первым делом производится вызов родительского конструктора при помощи ключевого слова inherited. Таким образом, унаследованный метод выполняет всю необходимую для на-

чальной инициализации объекта работу. Как нетрудно догадаться, это возможно благодаря такой концепции ООП, как полиморфизм.

Есть всего одно условие, касающееся расположения конструкторов и деструкторов в объявлении класса: они должны быть объявлены в общедоступной части, т.е. в public или в published.

Использование собственного конструктора необходимо в том случае, если объект в качестве своих полей содержит другие объекты. В таком случае следует написать собственный конструктор, который будет создавать эти объекты. Например, если в классе TUser было определено поле MegaInfo, являющееся объектом типа TMegaUserInfo, то конструктор класса TUser должен создавать соответствующий объект, вызывая его конструктор:

```
constructor TUser.Create;
begin
 inherited Create;
 MegaInfo:=TMegaUserInfo.Create;
 ...
end;
```

Что касается метода Destroy, то он призван освободить память, занимаемую экземпляром класса, и называется деструктором (destructor). Соответственно, в том случае, если среди полей класса будут другие объекты, использование переопределенного деструктора обязательно. Например, для класса TUser с полем типа TMegaUserInfo мы получим следующий деструктор:

```
destructor TUser.Destroy;
begin
 ...
 MegaInfo.Destroy;
 inherited Destroy;
end;
```

Обратите внимание на то, что если в конструкторе наследуемый метод Create всегда вызывается первым, то в деструкторе наследованный метод Destroy – последним.

Однако просматривая исходные коды других программ (включая библиотеку VCL) вы вряд ли обнаружите вызов метода Destroy напрямую. Дело в том, что к моменту уничтожения того же объекта TUser поле MegaInfo уже может быть освобождено, например, путем присвоения ему в программе значения «nil» (нулевого указателя):

```
User.MegaInfo:=nil;
User.Destroy; // здесь во время выполнения возникнет ошибка!
```

Таким образом, попытка повторно освободить память при помощи метода Destroy приведет к ошибке выполнения программы. Поэтому на практике, вместо метода Destroy всегда используют метод Free, который сначала проверяет, не является ли ссылка на объект нулевым указателем, и если нет – то только тогда вызывает деструктор. Соответственно, вид деструктора и его вызов из программы будут выглядеть таким вот образом:

```
destructor TUser.Destroy;
begin
...
```

```
MegaInfo.Free;
 inherited Destroy;
end;
...
User.MegaInfo:=nil;
User.Free; // теперь ошибки не будет
```

Таким образом, использование метода Free является прекрасной возможностью избежать непредвиденных ошибок времени выполнения.

Конструкторы и деструкторы, подобно другим подпрограммам, могут принимать значения в качестве параметров. Например, если при создании нового экземпляра класса TUser нам нужно было бы сразу получить его имя, то конструктор получился бы таким:

```
constructor TUser.Create(str: string);
begin
 inherited Create;
 fLogin:=str;
 fPassword:='password';
end;
```

Соответственно, создание объекта типа TUser в программе теперь будет выглядеть следующим образом:

```
User:=TUser.Create('Вася');
```

Аналогичным образом параметры можно передавать и деструкторам, при этом, в зависимости от значения параметров, они могут, скажем, выполнять или не выполнять перед уничтожением объекта какие-либо действия, вроде сохранения данных на диск, вывода того или иного сообщения и т.д.

Методы объектов и оператор with

Конструктор и деструктор вкупе с методом Free – далеко не единственные стандартные методы, достающиеся создаваемым классам в наследство от TObject. Рассмотрим здесь некоторые из них, которые могут быть полезными при работе с объектами в Delphi.

Начнем с того, что для конструктора и деструктора предусмотрены два дополнительных метода, которые вызываются автоматически сразу после создания объекта и непосредственно перед уничтожением. Это AfterConstruction и BeforeDestruction. Хотя к самим этим методам обращаться напрямую нельзя – они вызываются автоматически, допускается определить в них код, который следует выполнить после создания объекта, или перед его удалением.

Если в коде программы требуется выяснить, к какому классу принадлежит тот или иной объект, можно использовать метод ClassName, возвращающий имя класса в виде строки. А если требуется просто проверить, не является ли имя класса у объекта каким-либо конкретным значением, используют метод ClassNameIs:

```
str := User.ClassName; // в str получим имя класса, например, 'TUser'
if User.ClassNameIs('TUser') then ...
// условие будет верно, если объект User является экземпляром класса TUser
```

Для того же случая, если требуется получить ссылку именно на сам класс, а не его имя, предусмотрен метод ClassType. Однако напрямую его использовать не следует – для этих целей существует оператор is, который применим не только к простым типам, но и к классам:

```
if User is TUser then ...
// условие будет верно, если объект User является экземпляром класса TUser
```

Собственно говоря, другой оператор этой группы – as – так же применим к объектам. Например, если мы хотим использовать переменную User2, являющуюся экземпляром некого класса TCustomUser, в качестве объекта типа TUser, мы можем написать такое выражение:

```
(User2 as TUser).Login := 'Иван Иваныч';
(User2 as TUser).Password := 'пароль';
```

Такое выражение будет допустимо, если у того класса, которому принадлежит объект User2 (т.е. TCustomUser), так же имеются свойства Login и Password, причем они должны быть такого же типа, как и у TUser.

Рассмотренные методы иногда бывают полезными при работе с объектами, поскольку позволяют выполнить те или иные операции. А вот оператор with полезен скорее программисту, пишущему программу, поскольку в ряде случаев позволяет существенно сократить размер кода. Например, предыдущие выражения при помощи оператора with можно записать так:

```
With (User2 as TUser) do begin
 Login := 'Иван Иваныч';
 Password := 'пароль';
end;
```

Фактически, при помощи оператора with мы можем, единожды указав часть выражения до точки, далее его опускать до тех пор, пока не окончится блок with. При этом не важно, сколько уровней вложения будет использовано. Например, если взять класс TUser, который в качестве поля MegaInfo имеет класс TMegaUserInfo, а оно, в свою очередь, имеет поле Address, являющееся записью типа TAddress, то для обращения к полям записи всякий раз придется указывать довольно-таки длинные строки, определяющие расположение того или иного фрагмента данных:

```
User.MegaInfo.Address.PostIndex := '119071';
User.MegaInfo.Address.City := 'Москва';
```

Вместо этого, с использованием оператора with, можно лишь единожды написать весь «путь» к этим данным, в дальнейшем обращаясь лишь к самим полям:

```
With User.MegaInfo.Address do begin
 PostIndex := 119071;
 ...
end;
```

В операторе with допустимо использовать сразу несколько таких путей. Например, если надо будет обращаться и к полям записи Address, и непосредственно к полям User, то можно использовать такой синтаксис:

```
With User, User.MegaInfo.Address do begin
 PostIndex := 119071;
 Login := 'Иван Иваныч';
```

```
 MegaInfo.Phone := '123-4567';
 ...
end;
```

Здесь в первой строке будет использован второй путь – до полей Address, а во второй и третьей – первый, до полей объекта User. Попутно отметим, что оператор with может использоваться не только с классами, но и с обычными записями.

Классы в действии

Попробуем применить инкапсуляцию, наследование и полиморфизм в одном проекте, заодно воспользовавшись оператором with, и посмотрев, как все это работает. Для этого создадим несколько классов, причем один из них будет наследником другого. В качестве родительского класса используем TCustomUser, а класс TUser сделаем его наследником. Еще один класс – TMegaUserInfo будет являться полем класса TCustomUser. Наконец, одним из полей класса TMegaUserInfo пусть будет запись TAddress. В таком случае порядок объявления типов должен производиться в следующей последовательности: TAddress, TMegaUserInfo, TCustomUser, TUser. Таким образом, объявления типов на начальном этапе могут выглядеть таким образом, как показано в листинге 7.2.

Листинг 7.2. Черновое определение типов

```
type
 TAddress = record
   PostIndex: integer;
   City: string;
 end;
 TMegaUserInfo = class
   Address: TAddress;
   Phone: string;
 end;
 TCustomUser = class
  protected
    fRealName: string;
  public
    MegaInfo: TMegaUserInfo;
 end;
 TUser = class (TCustomUser)
  private
    fLogin: string;
    fPassword: string;
    procedure setPassword(newpass: string);
  published
    property Login: string read fLogin write fLogin;
    property Password: string read fPassword write setPassword;
end;
```

Что касается функционального содержимого классов, то для TCustomUser нам придется использовать собственный конструктор, поскольку одним из его полей является другой класс, из которого необходимо создать объект во время выполнения про-

граммы. Попутно конструктор может принимать исходное значение для одного из полей этого класса, скажем, для fRealName. А поскольку данное поле мы определили как защищенное, то неплохо было бы предусмотреть еще и общедоступный метод для извлечения этого значения. Это может быть свойство или функция. Выберем функцию и назовем ее GetRealName. Наконец, раз есть конструктор, создающий дополнительные объекты то нужен еще и деструктор, их уничтожающий. В результате для этого класса определение будет таким:

```
TCustomUser = class
  protected
    fRealName: string;
  public
    MegaInfo: TMegaUserInfo;
    function GetRealName: string;
    constructor Create(username: string);
    destructor Destroy; override;
 end;
```

Теперь займемся дочерним классом TUser. В дополнение к унаследованным от объекта TCustomUser полям и методам, он получит собственные. Пусть это будут свойства Login и Password, вместе с корреспондирующими защищенными переменными. Кроме того, предусмотрим для него конструктор, который будет устанавливать одно из свойств, скажем, Login, в какое-либо значение, определяемое при создании экземпляра класса. В результате мы получим следующее:

```
TUser = class (TCustomUser)
  private
    fLogin: string;
    fPassword: string;
    procedure setPassword(newpass: string);
  published
    property Login: string read fLogin write fLogin;
    property Password: string read fPassword write setPassword;
    constructor Create(username, userlogin: string);
 end;
```

Отметим, что по той причине, что конструктор этого класса должен вызывать конструктор родительского, то ему придется принимать сразу 2 параметра – для себя и для «родителя». Зато деструктор в данном случае нам не понадобится, поскольку очистку памяти, занимаемую полем-объектом MegaInfo, выполнит деструктор родительского класса, к которому данное поле и принадлежит. В этом мы убедимся по ходу выполнения программы. Для этого достаточно во всех методах этих классов предусмотреть вывод сообщения, рапортующего о ходе его вызова. Например, метод GetRealName будет выглядеть следующим образом:

```
function TCustomUser.GetRealName: string;
begin
 result:=fRealName;
 writeln(' >> Calls TCustomUser.GetRealName');
end;
```

В других методах мы так же разместим подобные инструкции для вывода сообщения об его исполнении и о полученных параметрах (разумеется, при их наличии). В результате заголовочная часть программы получит вид, показанный на листинге 7.3.

Листинг 7.3. Классы в полном составе

```
program class2;
{$APPTYPE CONSOLE}
type
 TAddress = record
   PostIndex: integer;
   City: string;
 end;
 TMegaUserInfo = class
   Address: TAddress;
   Phone: string;
 end;
 TCustomUser = class
  protected
    fRealName: string;
  public
    MegaInfo: TMegaUserInfo;
    function GetRealName: string;
    constructor Create(username: string);
    destructor Destroy; override;
 end;
 TUser = class (TCustomUser)
  private
    fLogin: string;
    fPassword: string;
    procedure setPassword(newpass: string);
  published
    property Login: string read fLogin write fLogin;
    property Password: string read fPassword write setPassword;
    constructor Create(username, userlogin: string);
 end;
function TCustomUser.GetRealName: string;
begin
 result:=fRealName;
 writeln(' >> Calls TCustomUser.GetRealName');
end;
constructor TCustomUser.Create(username: string);
begin
 inherited Create;
 MegaInfo:=TMegaUserInfo.Create;
 fRealName:=username;
 writeln(' >> Calls TCustomUser.Create with '+username);
end;
```

```
destructor TCustomUser.Destroy;
begin
 MegaInfo.Free;
 inherited Destroy;
 writeln(' >> Calls TCustomUser.Destroy');
end;
constructor TUser.Create(username, userlogin: string);
begin
 inherited Create(username);
 fLogin:=userlogin;
 fPassword:='password';
 writeln(' >> Calls TUser.Create with '+username+' and '+userlogin);
end;
procedure TUser.setPassword(newpass: string);
begin
 if Length(newpass)>3 then fPassword:=newpass
   else writeln('Error! Password is too short!');
 writeln(' >> Calls TUser.setPassword with '+newpass);
end;
var
 User: TUser;
begin
 ...
end.
```

В результате, если вызвать такой метод класса TUser, который имеет унаследованное представление (в нашем случае – конструктор), то в ходе выполнения программы будут выданы сообщения, показывающие порядок вызова подпрограмм. Например, при обращении к конструктору класса TUser можно будет отследить вызов конструктора класса TCustomUser:

```
User:=TUser.Create('Ivan Ivanovich','Vano');
```

В результате выполнения этой строки будут выданы 2 сообщения: «Calls TCustomUser.Create with Ivan Ivanovich» и «Calls TUser.Create with Ivan Ivanovich and Vano».

Полный код программы можно найти в каталоге Demo\Part2\Class2. В завершение отмечу, что это, пожалуй, будет последним консольным приложением среди примеров этой книги.

Визуальные компоненты и ООП

Все то, что было сказано о классах, является лишь прелюдией к основной составляющей Delphi – библиотеке визуальных компонент. Ну а поскольку компоненты VCL, в своей массе, рассчитаны именно на визуальное программирование, то с этого момента мы наконец-то перейдем к примерам качественно нового уровня, создаваемым в визуальной среде Delphi.

Библиотека VCL

Библиотека визуальных компонентов содержит множество классов, которые вы можете использовать в своих приложениях. Она написана на языке Object Pascal и непосредственно связана с интегрированной средой разработки Delphi. В частности, все кнопки, расположенные на палитре компонент являются ни чем иным, как представлением визуальных компонент VCL.

Вместе с тем, многие классы VCL не являются визуальными компонентами. Например, существует специальный класс TStringList, используемый для работы с группами строк, или же класс TRegistry, предназначенный для работы с реестром Windows. Но все классы в Delphi, как мы знаем, происходят от общего предка – класса TObject, а прочие классы являются его наследниками в том или ином поколении, образуя иерархию классов.

Следующим классом в иерархии классов VCL, после TObject, является класс TPersistent. В дополнение к методам TObject он имеет так же методы для присвоения данных свойствам и для обмена объектов данными между собой (метод Assign). Параллельно классу TPersistent существуют так же классы TException и TIniFile. Первый из них предназначен для работы с исключительными ситуациями (т.е. обеспечивает обработку ошибок времени выполнения), а второй инкапсулирует методы для работы с INI-файлами.

Если же говорить именно о компонентах, а не о каких-то абстрактных классах, то их прародителем является класс TComponent, являющийся прямым потомком класса TPersistent. Именно происходящие от TComponent классы являются компонентами Delphi, в том числе теми, что отображаются на палитре компонент. Помимо класса TComponent, у TPersistent имеются еще 2 потомка – класс TClipboard, предназначенный для работы с буфером обмена Windows и класс TStrings, являющийся основой для уже упоминавшегося класса TStringList.

При всем этом компоненты, происходящие от класса TComponent, не обязательно являются визуальными. Если же говорить именно о визуальных компонентах, то нам следует продвинуться еще дальше, к классу TControl, являющимся общим предком для всех элементов графического интерфейса в созданных при помощи Delphi приложениях Windows. Но реальные компоненты, как правило, происходят не от самого класса TControl, а от 2 его разновидностей, представленных, в случае для Windows, классами TWinControl и TGraphicControl. Отметим, что полноценными оконными элементами управления (с поддержкой ввода с клавиатуры, визуальной реакцией на действия пользователя и т.д.) являются только наследники класса TWinControl. Что касается компонент, происходящих от TGraphicControl, то они являются облегченным вариантом элементов интерфейса, не требующими поддержки всех функциональных возможностей управления со стороны операционной системы. Такими элементами являются, например, статические картинки, используемые в оформлении программ (вроде логотипа в окне About) и т.п.

Другими наследниками класса TComponent, являются такие классы, как TScreen и TApplication. Класс TScreen инкапсулирует в себе свойства и методы, необходимые для работы с экраном, на котором запущено приложение. А класс TApplication является основой для самого графического приложения, обеспечивая его взаимодействие

с операционной системой и выполняя рад иных вспомогательных действий вроде поддержки системного меню или перехвата и обработки нажатий клавиатуры.

Таким образом, если представить все рассмотренные здесь классы в виде некой древовидной структуры, выстроив ее по направлению от TObject до TWinControl, то мы получим фрагмент иерархии классов VCL, относящийся к визуальным компонентам и захватывающий некоторые ближайшие ответвления (рис. 8.1).

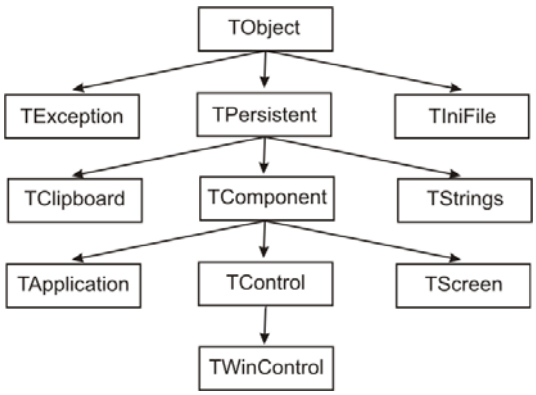

Рис. 8.1. Фрагмент иерархии классов VCL от TObject до TWinControl

Еще одним важным подспорьем в изучении VCL, помимо иерархии классов, является исходный код самой библиотеки VCL, поставляемый вместе с Delphi. Обычно они располагаются в каталоге Source, вложенном в тот каталог, в который вы установили саму Delphi, например, C:\Program Files\Borland\Delphi\Source.

Класс TComponent

Теперь настало время более подробно ознакомиться с классами, на которых основываются компоненты VCL. Начнем с основы – класса TComponent. В дополнение к методам своего родителя – класса TPersistent, он имеет ряд собственных методов, а также здесь впервые появляются свойства. Именно благодаря свойствам, определенным в этом классе, одни компоненты могут владеть другими. На практике это означает, что когда на такой компонент, как форма (окно), вы помещаете другой компонент, скажем, кнопку, то кнопка становится составной частью формы. Благодаря этой особенности при создании основного компонента, все принадлежащие ему компоненты будут созданы автоматически. Равно как и при удалении такого компонента, все принадлежащие ему компоненты так же удаляются автоматически.

Рассмотрим свойства класса TComponent более подробно, для чего обратимся к таблице 8.1

Таблица 8.1. Основные свойства TComponent

Свойство	Тип	Описание
ComponentCount	Integer	Число принадлежащих (дочерних) компонентов
ComponentIndex	Integer	Номер компонента в списке родительского компонента-владельца
Components	array of TComponent	Список всех принадлежащих компонентов

ComponentState	TComponentState	Указывает на текущее состояние компонента, может принимать значения csLoading, csReading, csWriting, csDestroying, csDesigning, csAncestor, csUpdating, csFixups, csFreeNotification, csInline, csDesignInstance
Name	String	Определяет имя компонента, по которому к нему следует обращаться из кода программы
Owner	TComponent	Ссылается на компонент, являющийся владельцем данного компонента
Tag	Integer	Хранит число, которое можно использовать в произвольных целях

Важно отметить, что многие из этих свойств доступны только для чтения, т.е. их нельзя изменить явно, присвоив им то или иное значение. Плный доступ допустим только для свойств Name, Tag и ComponentIndex. Кроме того, свойства Name и Tag являются опубликованными, и, соответственно, они будут видны в инспекторе объекта уже на этапе разработки программы.

Что касается методов, то их для класса TComponent определено предостаточно, однако подавляющее большинство из них используется либо для реализации каких-либо внутренних механизмов взаимодействия в VCL, либо имеет весьма специфическое предназначение. Что касается более приземленных методов, то среди них можно выделить группу из 4 методов, предназначенных для манипулирования принадлежащими компонентами:

- DestroyComponents – разрушает все принадлежащие компоненты
- FindComponent – возвращает ссылку на принадлежащий компонент по его имени, если такового не найдется, то возвращает nil;
- InsertComponent – добавляет компонент, указанный в качестве параметра, в качестве принадлежащего. При этом он заносится в самый конец списка Components;
- RemoveComponent – удаляет указанный в качестве параметра компонент из списка Components.

В качестве своего непосредственного родителя класс TComponent имеют такие компоненты, как таймер (TTimer), невизуальные компоненты для доступа к данным (TDataSet, TDataSource), компоненты для взаимодействия через DDE и другие. От него же ведут свою «родословную» многие основополагающие классы, на основе которых строятся другие компоненты. Самым интересным среди них, безусловно, является класс TControl, на основе которого построены все визуальные компоненты в Delphi. Именно на основе класса TControl можно изучить общие свойства всех элементов интерфейса (окон, кнопок, текстовых надписей, окон редактирования и т.д.), которые только можно создать в приложении.

Свойства визуальных компонент

Как уже было отмечено, все визуальные компоненты происходят от класса TControl. Всего для этого класса определено около 100 методов и свыше 50 свойств, и это – помимо свойств и методов, унаследованных от TComponent. Причем многие из них,

особенно если рассматривать свойства, довольно часто используются при практической разработке приложений в контексте того или иного визуального компонента, начиная от окна-формы, и заканчивая текстовыми подписями.

Рассмотрим такие свойства, обратившись к таблице 8.2.

Таблица 8.2. Основные свойства класса TControl

Свойство	Тип	Описание
Action	TBasicAction	Определяет действие, связанное с данным элементом
Align	TAlign	Определяет выравнивание элемента относительно родительского элемента. Допустимые значения: alNone, alTop, alBottom, alLeft, alRight, alClient, alCustom
Anchors	TAnchors	Определяет, по каким краям данный элемент привязывается к родительскому. Допустимые значения akTop, akLeft, akRight, akBottom
AutoSize	Boolean	Определяет, может ли размер элемента изменяться автоматически, в зависимости от содержимого
Caption	Sttring	Определяет текстовую строку, идентифицирующую элемент для пользователя
ClientHeight	Integer	Определяет высоту внутреннего пространства элемента, доступную для размещения других элементов в пикселях
ClientWidth	Integer	Определяет ширину внутреннего пространства элемента, доступную для размещения других элементов в пикселях
Color	TColor	Определяет цвет фона элемента. Цвет задается при помощи ключевых слов (clNone, clAqua и т.д.) или числовым значением в диапазоне -$7FFFFFFF-1..$7FFFFFFF;
Constraints	TSizeConstraints	Определяет минимальные и максимальные размеры элемента, задаваемые четырьмя целыми
Cursor	TCursor	Определяет тип курсора. Например, crDefault, crNone и т.д.
DragCursor	TCursor	Определяет тип курсора в момент перетаскивания. Например, crDefault, crNone и т.д.
DragKind	TDragKind	Определяет, будет ли элемент при перетаскивании просто перемещаться или должен быть пристыкован, может принимать значения dkDrag, dkDock
DragMode	TDragMode	Определяет, как должны инициироваться операции drag-and-drop и drag-and-dock. Допустимые значения dmManual, dmAutomatic
Enabled	Boolean	Определяет, должне ли элемент реагировать на события клавиатуры, мыши и таймера
Font	TFont	Определяет атрибуты шрифта для текстового наполнения элемента.
Height	Integer	Определяет высоту элемента в пикселях
HelpContext	Integer	Определяет индекс раздела в файле справки, относящегося к этому элементу
HelpKeyword	String	Определяет ключевое слово в файле справки, относящегося к этому элементу

HelpType	THelpType	Определяет, по какому критерию должно производиться обращение к файлу справки, по слову (htKeyword), или по номеру (htContext)
Hint	String	Определяет текст всплывающей подсказки
Left	Integer	Определяет горизонтальную координату левой стороны элемента в пикселях по отношению к родительскому
Name	String	Определяет имя элемента
Parent	TWinControl	Определяет родительский элемент, в целом аналогична свойству Owner, но всегда ссылается на элемент типа TWinControl
ParentColor	Boolean	Определяет, должен ли элемент унаследовать фоновый цвет у содержащего объекта
ParentFont	Boolean	Определяет, должен ли элемент унаследовать оформление шрифта у содержащего объекта
ParentShowHint	Boolean	Определяет, должен ли элемент унаследовать показ подсказки у содержащего объекта
PopupMenu	TPopupMenu	Определяет всплывающее (контекстное) меню, связанное с данным элементом
ShowHint	Boolean	Определяет, показывать или нет всплывающую подсказку для данного элемента
Text	String	Определяет текст, который должен быть отображен элементом. В зависимости от типа элемента, используется либо свойство Text, либо Caption
Top	Integer	Определяет вертикальную координату верхней стороны элемента в пикселях по отношению к родительскому
Visible	Boolean	Определяет, должен ли компонент быть видимым
Width	Integer	Определяет ширину элемента в пикселях

Достаточно создать новый проект в Delphi (File → New → Application), и щелкнуть по окну первой формы, созданной для приложения (Form1), чтобы увидеть многие из перечисленных в таблице свойств в окне инспектора объекта. Разумеется, учитывая то, что класс TForm, на основе которого создаются формы, является наследником рассматриваемого класса TControl далеко не в первом поколении, а на каждом этапе добавляются все новые и новые свойства, то список доступных свойств будет большим, нежели в таблице.

Между тем, из приведенных здесь свойств, некоторых может и не оказаться среди перечня свойств того или иного компонента. Это объясняется тем, что они скрываются за ненадобностью. Например, для формы не предусмотрено такого свойства, как DragCursor, поскольку для окна оно бессмысленно – как известно, операции типа «перетащи и брось» в Windows с окнами не производятся.

С теми или иными исключениями и дополнениями данные свойства имеются и для других визуальных компонент – будь то кнопки, поля редактирования текста, списки, рамки, переключатели и т.д. Причем общее их количество довольно сильно варьируется в зависимости от конкретного компонента. Свою долю вносит и версия Delphi, поскольку от версии к версии могут добавляться какие-либо новые свойства. В результате для Delphi 7 мы имеем 60 свойств для формы, 33 – для кнопки и 47 – для поля редактирования текста (рис. 8.2).

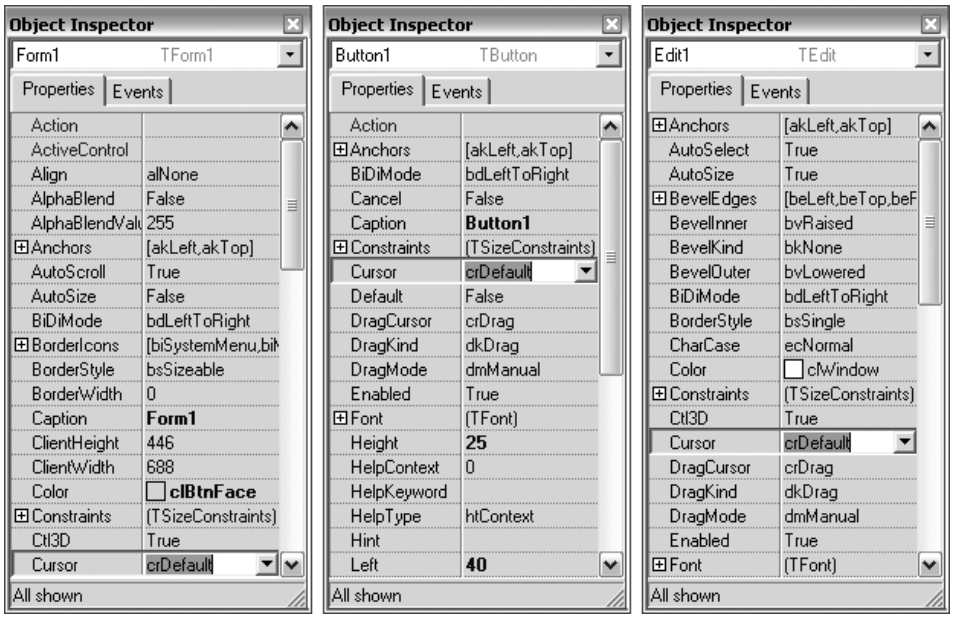

Рис. 8.2. Инспектор объекта со свойствами формы, кнопки и текстового поля

Из тех свойств, что всегда доступны для любого объекта в VCL, можно выделить такие, как Tag и Name. Если же брать визуальные компоненты, то для них всегда актуальны так же свойства, отвечающие за размеры (Height, Width и т.д.), вид курсора, шрифт, связь со справочной системой и т.д.

СОВЕТ

> Всегда можно получить подсказку по любому интересующему вас свойству, выбрав его в инспекторе объекта и нажав на клавишу F1.

Что касается методов, предоставляемых классом TControl, то, с одной стороны, их еще больше, чем свойств, а с другой – сама суть их использования еще более зависима от того, в контексте какого компонента тот или иной метод используется. По этой причине мы будем рассматривать их постепенно, преимущественно по мере ознакомления с конкретными компонентами.

События

Вся работа Windows построена вокруг событий. Событием является нажатие на клавиатуру, движения и щелчки мышки, действия внутренних устройств компьютера и т.д. Причем каждое событие генерирует сообщение, которое необходимо перехватить и обработать. При этом операционная система производит предварительную сортировку всех этих сообщений, и направляет приложению только ту их часть, которая к нему относится. И все равно таких сообщений множество, поскольку даже если пользователь всего лишь проведет указателем мышки над окном приложения, это инициирует целый поток сообщений.

К счастью, Delphi большую часть работы по обработке событий берет на себя. Фактически, при разработке приложений, использующему Delphi разработчику необходимо всего лишь установить собственные обработчики на те события, которые ему действительно необходимо обработать. Типичный пример – нажатие на клавишу мышки. Причем, опять-таки, благодаря Delphi, не понадобиться отслеживать различные фазы нажатия (кнопка опущена – кнопка отжата – кнопка отпущена), как и следить за тем, от какого источника пришел сигнал (т.е. был ли это щелчок мышкой, или нажатие Enter, или было использовано сочетание горячих клавиш).

Отметим, что события, как часть класса VCL, впервые появляются, начиная как раз с класса TControl. Таким образом, в дополнение к полутора сотням свойств и методов этого класса, мы получим еще некоторое количество событий. Собственно говоря, событие представляет собой ничто иное, как свойство процедурного типа, предназначением которого является обеспечение реакции на те или иные действия. Когда такому свойству присваивают значение, то фактически указывается метод, вызываемый при наступлении данного события. Подобные методы называют обработчиками событий (event handlers).

События в Delphi подразделяются на несколько типов, в зависимости от того, что представляет собой то или иное событие. Простейшим и одним из наиболее распространенных является тип TNotifyEvent, характерный для обычных уведомляющих событий. В VCL этот тип описан следующим образом:

```
type TNotifyEvent = procedure(Sender: TObject) of object;
```

Единственный параметр Sender здесь обозначает объект, являющийся источником возникшего события.

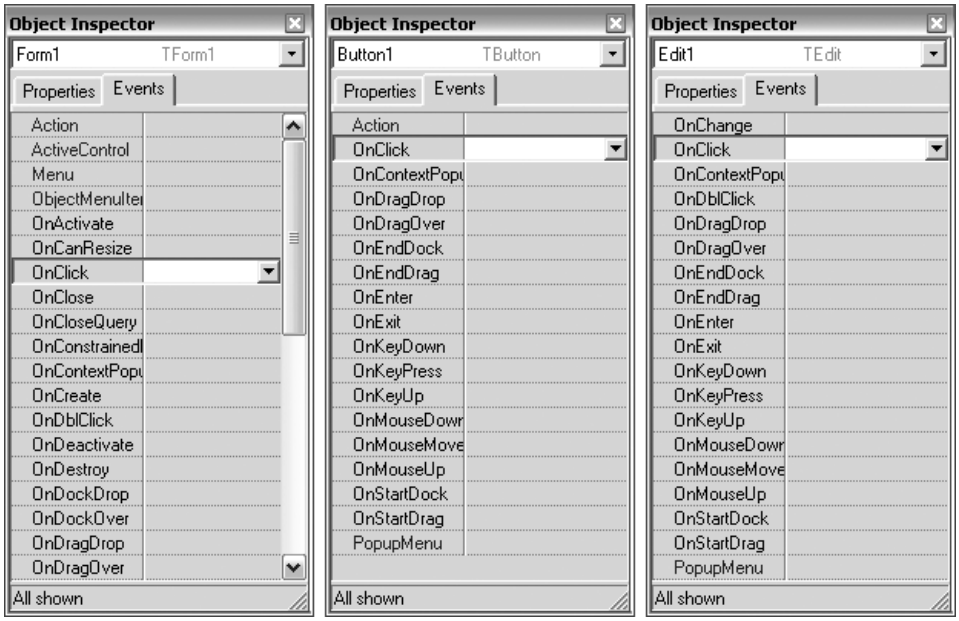

Рис. 8.3. Инспектор объекта со списком событий для формы, кнопки и текстового поля

Переходя от общих слов к делу, вновь обратим свое внимание на инспектор объекта. Можно заметить, что помимо закладки свойств (properties), он содержит еще и вторую закладку – события (events). На ней приводится список событий, которые могут быть обработаны данным объектом (рис. 8.3).

Подобно свойствам, многие события повторяются для разных объектов. Наиболее распространенным типом является событие OnClick, которое происходит в момент щелчка мышкой – оно существует практических у всех визуальных компонент.

Для того чтобы назначить обработчик событию, при разработке приложения в среде Delphi, достаточно сделать двойной щелчок по полю нужного события в инспекторе объекта. Например, если выбрать форму и дважды щелкнуть по строке инспектора объектов напротив надписи «OnClick», то Delphi автоматически сгенерирует весь нужный код. Прежде всего, это будет заготовка для самого обработчика событий:

```
procedure TForm1.FormClick(Sender: TObject);
begin


end;
```

Кроме того, если теперь посмотреть объявление самой формы, то можно будет увидеть, что в часть объявлений класса TForm1 добавилось объявление этой процедуры:

```
procedure FormClick(Sender: TObject);
```

Таким образом, нам остается лишь добавить свой код в тело процедуры, созданной для обработчика события щелчка мышкой по форме. Пусть это будет смена заголовка окна с предлагаемого по умолчанию значения (Form1) на все ту же фразу «Hello, World!». Для этого достаточно обратиться к свойству Caption объекта Form1, и назначить ему новое значение. В итоге, вся процедура примет следующий вид:

```
procedure TForm1.FormClick(Sender: TObject);
begin
 Form1.Caption:='Hello, World!';
end;
```

Теперь, если запустить такое приложение на выполнение (нажав F9), и щелкнуть в любом месте его окна мышкой, то текст заголовка окна изменится с «Form1» на «Hello, World!».

Методы визуальных компонент

Итак, события представляют собой свойства, для которых определяются методы. Таким образом, от событий мы вновь возвращаемся к теме методов, в аспекте их работы с визуальными компонентами.

ПРИМЕЧАНИЕ

> Отметим, что в отличие от обычных свойств, или от событий, для методов в инспекторе объектов места не нашлось. Поэтому чтобы получить справочную информацию о методах, следует сперва обратиться к справке по самому компоненту, а уже из самого окна справки щелкнуть по ссылке «Methods».

Прежде всего, методы предоставляют иной путь управления компонентами. Поскольку компоненты – это все те же объекты, они могут иметь процедуры и функции.

Таким образом, по аналогии с любыми другими классами, метод визуального компонента представляет собой процедуру или функцию, объявленную в общедоступной части класса, и которую можно вызвать при работе с данным компонентом. Например, когда мы изменяли заголовок окна, то использовали свойство Caption. А вот если бы мы хотели, скажем, скрыть форму (сделать ее невидимой), то следовало бы обратиться к методу Hide:

```
Form1.Hide;
```

Помимо метода Hide, а так же определенных еще для родительских классов методов вроде Create и Free, класс TControl имеет ряд других часто применяемых методов, которые наследуются визуальными компонентами-потомками этого класса. Некоторые из наиболее часто употребляемых методов приведены в таблице 8.3.

Таблица 8.3. Некоторые методы класса TControl

Метод	**Описание**
BringToFront	Делает форму видимой или переносит элемент, скрытый на форме под другими элементами, на передний план
Hide	Делает элемент невидимым
Perform	Пересылает элементу указанное Windows-сообщение
Refresh	Инициирует обновление (перерисовку) объекта на экране
Repaint	Синоним для Refresh
SendToBack	Переносит элемент на задний план
Show	Делает элемент видимым
Update	Инициирует немедленное выполнение всех сообщений, связанных с ожиданием обновления (перерисовки) элемента

Важно отметить, что большинство компонент, в отличие от обычных классов, нельзя создать «просто так». Т.е. каждый компонент должен иметь своего родителя, к которому он привязан. Поэтому если требуется создать, скажем, кнопку, причем именно программным методом, то недостаточно просто создать экземпляр класса – необходимо так же указать его родительский объект, для чего, во-первых, уже при обращении к конструктору указывают имя объекта-контейнера, а во-вторых, уже после создания явно определяют свойство Parent. В качестве родительского компонента может выступать любой другой компонент, который может содержать в себе иные компоненты, например, форма:

```
var
  Form1: TForm1;
  But: TButton;
...
But:=TButton.Create(Form1);
But.Parent:=Form1;
```

К счастью, при создании экземпляров компонент визуальными методами (т.е. в графической среде Delphi) вся необходимая работа выполняется автоматически.

Типы методов

Метод может вызываться различными способами, в зависимости от вида самого метода. Все методы подразделяются на несколько видов – статические, динамические,

виртуальные и т.д. Тип метода задается при его описании в классе при помощи одного из ключевых слов:

- static – статический метод, такой тип методы имеют по умолчанию. Статические методы нельзя переопределять, но можно делать перегружаемыми;
- virtual – виртуальный метод. Виртуальные методы можно переопределять, а их выполняемый код оптимизируется под скорость выполнения;
- dynamic – динамический метод. Аналогичен виртуальному методу с той лишь разницей, что его исполняемый код будет оптимизирован по размеру;
- override – переопределяемый метод. Директива override указывает, что данный метод переопределяет унаследованный одноименный виртуальный или динамический метод;
- overload – перегружаемый метод. Перегружаемые методы не скрывают наследуемые методы, при этом они должны различаются по своим параметрам;
- message – метод обработки сообщения.

Для примера переопределяемого определим некий класс TFigure с методом Draw, а так же создадим его потомков – классы TCircle и TSquare с одноименными методами, которые и переопределим:

```
type
 TFigure = class
   procedure Draw; virtual;
 end;
 TCircle = class(TFigure)
   procedure Draw; override;
 end;
 TSquare = class(TFigure)
   procedure Draw; override;
 end;
```

Если теперь создать переменную типа TFigure, и использовать конструктор класса TSquare, а затем вызвать метод Draw, то вызван будет метод класса TSquare. Если же для переменной использовать конструктор класса TCircle, то метод уже этого класса будет вызван:

```
var
  Fig: TFigure;
begin
  Fig := TSquare.Create;
  Fig.Draw;  // вызов TSquare.Draw
  Fig.Destroy;
  Fig := TCircle.Create;
  Fig.Draw;  // вызов TCircle.Draw
  Fig.Destroy;
end;
```

В то же время, если бы, скажем, класс TCircle не имел собственного определения метода Draw, то был бы вызван метод, унаследованный от класса TFigure. Что каса-

ется перегружаемых методов, то суть их использования сводится к тому, что на разных уровнях наследования в родственных классах можно использовать одноименные методы, различающиеся лишь количеством и (или) типом параметров. Например, если взять те же классы TFigure и TSquare, то можно было бы использовать не переопределяемые, а перегружаемые методы:

```
type
TFigure = class
   procedure Draw(left, top: integer);
 end;
 TCircle = class(TFigure)
   procedure Draw(left, top, radius: integer); overload;
 end;
 TSquare = class(TFigure)
   procedure Draw(left, top, width, height: integer); overload;
 end;
```

Таким образом, в зависимости от того, с какими параметры будет вызван метод Draw, будет произведено обращение к той или иной процедуре:

```
var
  Fig: TFigure;
begin
  Fig := TSquare.Create;
  Fig.Draw(10,10,30,40);  // вызов TSquare.Draw
  Fig.Draw(5,50);  // вызов TFigure.Draw
  Fig.Destroy;
end;
```

Следует так же отметить, что если перегружается виртуальный метод, то в дочерних классах следует использовать директиву reintroduce. Наконец, полезно так же знать, что перегружаемыми могут быть не только методы классов, но и любые другие процедуры и функции.

Черчение, рисование и печать

Среду Windows не зря называют графической – весь текст, рисунки и элементы управления выводятся в графическом виде. Соответственно весь вывод приложений Windows, в отличие от рассмотренных ранее консольных программ, является графическим по своей природе.

Графика в Windows

Когда приложению требуется вывести какую-либо информацию на экран дисплея, операционная система – Windows – предоставляет ей в свое распоряжение оконно-ориентированную графику. Это означает, что каждая форма (как, впрочем, и ряд других объектов) рассматривается как поверхность для рисования – холст (canvas).

Когда в форме производится построение рисунка, то выводимое изображение обрезается по ее краям, т.е. выводятся только те его части, которые помещаются на холст окна. Такой подход гарантирует, что вывод одного приложения не затронет клиент-

ской области других приложений, на чем, собственно и основывается оконная система Windows.

За точку отсчета координат в окнах принимается левый верхний угол его клиентской области, ограниченной рамкой (т.е. как раз те габариты, что отражаются свойствами ClientHeight и ClientWidth). Для других элементов, имеющих холст для рисования, подобной рамки может и не быть – в таком случае рамками такого виртуального холста выступают границы объекта.

Вообще же прямое обращение к графическим функциям требуется в тех случаях, когда требуется создать какой-либо специфический элемент управления, не предусмотренный стандартными компонентами. В таких случаях всю заботу о своевременной перерисовке, визуальной реакции на действия пользователя и т.д., разработчику требуется взять на себя, предусмотрев весь необходимый для этого программный код. Разумеется, это довольно-таки кропотливая работа, особенно если создаваемый компонент дожжен обеспечивать интерактивность на должном уровне.

Хорошая новость состоит в том, что Delphi имеет целый ряд готовых к использованию компонент, заботиться о рисовании которых разработчику совершенно незачем, поскольку всю рутинную работу уже выполнили разработчики Delphi и VCL. Более того, в VCL есть множество «заготовок», на основании которых можно создавать собственные компоненты, опять-таки без непосредственного обращения к функциям графического вывода.

В то же время, рисование может быть полезным и при более простых ситуациях, например, когда требуется именно вывести какие-либо графические примитивы, или даже дать пользователю возможность что-нибудь начертить. Наконец, мы рассмотрим здесь вопрос рисования, чтобы еще раз проиллюстрировать работу классов и использование ООП, а заодно исследовать еще один фрагмент VCL.

Объект Canvas

Для многих элементов интерфейса в Windows-приложениях используется компоненты, основанные на классе TWinControl, являющегося прямым наследником уже рассмотренного нами класса TControl. Но множество других визуальных компонент происходят от другого его наследника – класса TGraphicControl, являющийся более простым и имеющим такое свойство, как Canvas, при помощи которого можно в буквальном смысле рисовать в Windows. Собственно говоря, свойство Canvas имеется не только у этого класса, но и у ряда других, в том числе и основанных на TWinControl. Впрочем, в любом случае это свойство будет представлено одним и тем же классом – TCanvas.

Объект Canvas инкапсулирует в себе взаимодействие с Windows GDI (Graphic Device Interface – интерфейс графического устройства). GDI обрабатывает весь графический вывод, предназначенный для экрана монитора, а так же для принтеров и иных печатающих устройств. Но поскольку взаимодействие с GDI, как, впрочем, и с другими подсистемами Windows API напрямую – дело хлопотное и чреватое ошибками, то в Delphi и был предусмотрен класс TCanvas, предоставляющий сравнительно простой и удобный доступ к GDI.

Сам по себе холст (Canvas) представляет собой поверхность, на которую можно выводить текст и иные графические изображения – линии, прямоугольники, эллипсы и

т.д. Но при ближайшем рассмотрении можно увидеть, что этот холст представляет собой плоскость, состоящую из отдельных точек – пикселей. Пиксель – это базовый элемент графического ввода, представляющий собой отдельную точку. Фактически, при рисовании на холсте вы просто закрашиваете его отдельные точки тем или иным цветом. Но, разумеется, работая с холстом посредством методов, предоставляемых классом TCanvas, можно без лишних хлопот выводить не только точки, но и текст, линии, прямоугольники, многоугольники, окружности, и даже готовые изображения. Рассмотрим основные свойства и методы объекта Canvas, обратившись к таблице 9.1.

Таблица 9.1. Свойства и методы TCanvas

Свойство (метод)	Тип значений или параметры	Описание
Pixels	Матрица TColor	Предоставляет доступ к любому пикселю холста, чтобы узнать или изменить его цвет
Pen	TPen	Свойства пера для черчения линий
Brush	TBrush	Свойства кисти для заполнения внутренних областей фигур
Font	TFont	Свойства шрифта для вывода текста
MoveTo	X, Y: Integer	Устанавливает текущую позицию пера
LineTo	X, Y: Integer	Проводит линию от текущей позиции к указанной
TextOut	X, Y: Integer; const Text: string	Выводит заданный текст, начиная с указанных координат
Rectangle	X1, Y1, X2, Y2: Integer или Rect: TRect	Рисует прямоугольник указанных размеров. Цвет рамки определяется текущим значением свойства Pen, а цвет заливки – свойством Brush
Ellipse	X1, Y1, X2, Y2: Integer или Rect: TRect	Рисует эллипс, вписанных в прямоугольник указанных размеров. Цвет рамки определяется текущим значением свойства Pen, а цвет заливки – свойством Brush
Polygon	Points: array of TPoint	Рисует многоугольник по указанным вершинам
PolyLine	Points: array of TPoint	Рисует ломаную линию, соединяющую указанные точки
Draw	X, Y: Integer; Graphic: TGraphic	Выводит графическое изображение, начиная от указанных координат (левого верхнего угла)

Рассмотрим некоторые свойства более подробно. Начнем с Pixels. Это свойство, представляющее собой двумерный массив, содержит информацию о цвете каждой точки поверхности, описывая, таким образом, всю поверхность холста. Изменяя цвет той или иной точки, мы можем выводить изображения. Например, если нам надо изменить цвет точки в левом верхнем углу, мы можем написать такое выражение:

```
Canvas.Pixels[1,1]:=clRed;
```

Здесь мы обратились к 1-му элементу 1-го ряда массива, которой как раз и соответствует левому верхнему углу холста, и назначили ему значение clRed, т.е. установили красный цвет.

Поскольку с использованием различных методов объекта Canvas можно выводить не только точки, но и различные фигуры, а так же текст, то остальные свойства как раз позволяют настроить параметры вывода этих фигур и текста. Так, для текста используется свойство Font, являющееся, в свою очередь, классом TFont и имеющим такие свойства, как Color (цвет), Name (гарнитура шрифта), Size (размер) и Style (стиль).

Последнее свойство имеет 4 флага, позволяющих сделать шрифт полужирным, наклонным, подчеркнутым или зачеркнутым:

```
Canvas.Font.Color:=clBlue; //шрифт будет синего цвета
Canvas.Font.Name:='Arial'; //выбрана гарнитура Arial
Canvas.Font.Size:=12; //установлен размер шрифта в 12pt
Canvas.Font.Style:=[fsBold,fsItalic]; //шрифт будет полужирным наклонным
```

Что касается таких свойств холста, как Pen и Brush, то для них предусмотрено изменение цвета и стиля линии (для Pen) или заполнения (для Brush). Кроме того, для пера можно определить ширину линии и режим наложения цвета. Соответственно, мы имеем следующие свойства:

- Color – определяет цвет линии для пера или заполнения для кисти;
- Style – определяет стиль линии или заливки. Для линии возможны следующие значения: psSolid, psDash, psDashDot, psDashDotDot, psClera и psInsideFrame. Для заливки: bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross;
- Width – определяет толщину линии в пикселях.

Следует учитывать, что изменить стиль линии возможно только в том случае, если ее толщина установлена в 1 пиксель (что, впрочем, является значением по умолчанию). При любом другом значении толщины линия всегда будет сплошной – psSolid.

Черчение фигур

Чтобы лучше представить себе использование объекта Canvas, попробуем использовать его свойства и методы для рисования фигур. Для начала попробуем вывести простую линию на поверхность формы, для чего создадим новое приложение, щелкнем по автоматически созданной форме (Form1), после чего перейдем в окно инспектора объекта и на закладке Events дважды щелкнем по строке напротив надписи OnClick. В ответ на это Delphi создаст обработчик события FormClick в редакторе кода:

```
procedure TForm1.FormClick(Sender: TObject);
begin


end;
```

Теперь остается поместить в него код, рисующий линию. Поскольку у формы уже имеется свойство Canvas, то будет достаточно просто обратиться к его методам MoveTo и LineTo. Пусть линия начнется в точке со смещением в 10 пикселей от левого верхнего угла по вертикали и горизонтали, и продлится до точки в 200 пикселей по горизонтали. В результате код получится следующим:

```
procedure TForm1.FormClick(Sender: TObject);
begin
 Form1.Canvas.MoveTo(10,10);
 Form1.Canvas.LineTo(200,10);
end;
```

Теперь остается запустить приложение и щелкнуть по любому мету на форме. Результатом будет вывод тонкой горизонтальной линии. Добавив к коду процедуры вызов метода Ellipse, мы получим построение эллипса, а Rectangle - прямоугольника:

```
Form1.Canvas.Ellipse(30,30,150,150);
Form1.Canvas.Rectangle(230,30,350,150);
```

Здесь в обоих случаях мы построили правильные фигуры, т.е. окружность и квадрат. Но поскольку для класса TCanvas не определены методы, строящие именно эти фигуры, то мы использовали методы для построения эллипса и прямоугольника, рассматривая окружность и квадрат как частные случаи этих типов фигур. В то же время, при необходимости можно было бы создать собственные методы, добавив их к классу TCanvas. Сделать это, на самом деле, несложно: достаточно определить новый класс, являющийся наследником TCanvas, и определить для него 2 новых метода. Назовем такой класс TMyCanvas, а методы – Circle и Square:

```
TMyCanvas = class(TCanvas)
  procedure Circle(Rad, X, Y: integer);
  procedure Square(Size, X, Y: integer);
end;
```

Определение этого класса следует поместить в части interface, непосредственно после определения класса TForm1, которое уже размещено в модуле самой Delphi. В части же implementation мы определим сами эти функции:

```
procedure TMyCanvas.Circle(Rad, X, Y: integer);
begin
end;
procedure TMyCanvas.Square(Size, X, Y: integer);
begin
end;
```

Теперь остается написать код, выполняющий построение фигур. Для начала примем, тот факт, что для метода Circle параметр Rad означает радиус окружности, а параметры X и Y – ее центр. Таким образом, мы можем использовать унаследованный метод Ellipse, подставив нужные параметры в его вызов:

```
Ellipse(X-Rad, Y-Rad, X+Rad, Y+Rad);
```

Что касается метода Square, то его параметр Size будет означать размер стороны квадрата, а X и Y – координаты верхнего левого угла. Таким образом, можно использовать метод Rectangle, указав для него соответствующие параметры:

```
Rectangle(X, Y, X+Size, Y+Size);
```

Теперь, когда методы определены и новый класс готов, следует разобраться с тем, как его использовать. Прежде всего, нам понадобится определить переменную - экземпляр класса. Кроме того, нам понадобится создать ее, использовав конструктор Create. Ну и, наконец, используя свойство Handle (указатель), мы должны связать наш объект с холстом формы. Все это можно разместить во все том же методе обработки щелчка мышкой.

```
procedure TForm1.FormClick(Sender: TObject);
var
 MC: TMyCanvas;
begin
 MC:=TMyCanvas.Create; // используем конструктор родительского класса
 MC.Handle:=Canvas.Handle; // назначаем холст окна областью вывода
 MC.Circle(50,200,100); // рисуем окружность диаметром 50 пикселей
 MC.Square(50,100,100); // рисуем квадрат со сторонами 50 пикселей
```

```
 MC.Free;
end;
```

Разумеется, все методы, доставшиеся классу TMyCanvas в наследство от TCanvas, так же можно использовать, включая те же Create и Free, чем мы и воспользовались. Применительно к остальным свойствам и методам можно делать то же самое, например, установить толщину и цвет линии, вывести линию при помощи MoveTo и т.д.:

```
MC.Pen.Width:=3;
MC.Pen.Color:=clRed;
MC.MoveTo(10,10);
MC.LineTo(200,10);
MC.Brush.Style:=bsHorizontal;
```

Таким образом, можно оформить вывод фигур, при помощи заполнения, штриховкой, а линию сделать толще и изменить ее цвет (рис. 9.1).

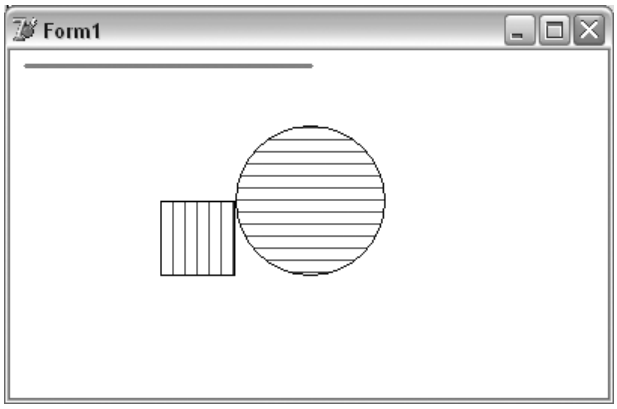

Рис. 9.1. Вывод графики на холст окна

Законченный код программы можно найти в каталоге Demo\Part2\Canvas. В нем же приведен еще один вариант использования метода Circle – для создания кольца из окружностей.

Вывод на печать

Вывод текста и графики возможен не только на дисплей, но и на принтер. Для этого так же используется свойство Canvas, но только не формы, а специального объекта Printer. Чтобы задействовать этот объект, необходимо подключить к программе модуль printers, указав его в списке используемых модулей, т.е. в uses. При этом не требуется ни создавать экземпляр класса TPrinter, ни заботиться о его удалении – достаточно просто использовать переменную Printer, которая создается для программы автоматически, если указан модуль Printers.

Помимо свойства Canvas, объект Printer имеет ряд иных свойств и методов, необходимых для осуществления печати. Среди методов этого объекта следует отметить, прежде всего, BeginDoc и EndDoc. Эти методы используются, соответственно, для начала и окончания процесса печати. Отметим так же метод NewPage, используемый для обозначения начала новой страницы, а так же метод Abort, прерывающий процесс печати. Что касается свойств, то они приведены в таблице 9.2.

Таблица 9.2. Свойства класса TPrinter

Свойство	Тип	Описание
Aborted	Boolean	Указывает, остановлена или нет печать пользователем
Canvas	TCanvas	Представляет холст листа
Copies	Integer	Определяет число копий для печати
Orientation	TPrinterOrientation	Определяет формат расположения бумаги – портретный (poPortrait), или ландшафтный (poLandscape)
PageHeight	Integer	Указывает на высоту страницы в пикселях
PageNumber	Integer	Указывает на текущую печатаемую страницу
PageWidth	Integer	Указывает на ширину страницы в пикселях
Printing	Boolean	Указывает, выполняется или нет печать в данный момент
Title	String	Определяет текст, идентифицирующий задание печати в менеджере принтеров Windows

Прежде, чем что-либо выводить не печать, необходимо проделать подготовительную работу. Это вызвано рядом причин, в частности, тем, что принтер не может печатать под обрез – следовательно, надо определить поля. Кроме того, у разных принтеров различное выходное разрешение, которое зависит как от модели, так и от настроек качества печати. В случае вывода текста так же следует определиться с количеством строк, помещающихся на странице. Тем не менее, в простейшем случае для вывода на печать бывает достаточно просто открыть документ для печати при помощи метода BeginDoc и назначить свойству Canvas принтера необходимое содержимое:

```
Printer.BeginDoc;
Printer.Canvas.TextOut(150, 150, 'Текст для печати');
Printer.EndDoc;
```

Однако если текст окажется слишком длинным, то он не перенесется на следующую строку, а просто не будет напечатан. Поэтому для вывода достаточно большого по объему текста, все-таки придется заниматься расчетами и готовить строки к выводу.

В то же время, если надо вывести только текст, причем его оформление не принципиально, то можно воспользоваться стандартной процедурой writeln, указав для нее вывод на принтер. Делается это при помощи процедуры AssignPrn:

```
var
 F: TextFile;
...
AssignPrn(F);
Rewrite(F);
writeln(F,'Текст для печати.');
CloseFile(F);
```

Разумеется, этот путь проще, в частности перенос длинных строк будет выполняться автоматически, но и недостатков у него хватает: мало того, что таким образом невозможно выводить графику, но и шрифт допускается использовать только один, без возможности изменять даже его размер или начертание в процессе вывода документа на печать.

Чтобы посмотреть, как работают оба этих метода, можно обратиться к примеру, находящемуся в каталоге Demo\Part2\Print.

Но вернемся к графическому выводу на принтер. Как уже было отмечено, для начала следует определиться с областью, доступной для печати и с ее размерами в пикселях. Для удобства бывает полезным определить переменные, отвечающие за размер полей с каждой из сторон документа. В случае вывода строк текса так же необходима переменная, определяющая положение ввода текущей строки. Что касается предмета для вывода, то попробуем нарисовать рамку для всей страницы, проходящую по ее полям, а так же текст разного размера и начертания, и какую-либо еще геометрическую фигуру. Все это мы можем разместить в функции, которую назовем PrintAny (см. листинг 9.1).

Листинг 9.1. Печать текста и графики

```
procedure PrintAny;
var
 LMargin, RMargin, TMargin, BMargin: integer;
 fH, HPos: integer;
begin
 with Printer, Printer.Canvas do begin
   LMargin:=PageWidth div 10; // поле слева 10% от ширины страницы
   RMargin:=PageWidth div 20; // поле справа 5% от ширины страницы
   TMargin:=PageHeight div 20; // поле сверху 5% от высоты страницы
   BMargin:=PageHeight div 10; // поле снизу 10% от высоты страницы
   Title:='Пробная печать'; // заголовок печати
   BeginDoc;
   Pen.Width:=3;
   Rectangle(LMargin,TMargin,PageWidth-RMargin,PageHeight-BMargin); // рамка
   Font.Name:='Arial';
   Font.Style:=[fsBold,fsUnderline];
   Font.Size:=24;
   fH:=abs(Font.Height); // получаем высоту шрифта
   HPos:=TMargin+fH; // вычисляем отступ сверху для вывода текста
   TextOut(LMargin,Hpos,' Заголовок ');
   Font.Style:=[];
   Font.Size:=10;
   fH:=abs(Font.Height);
   HPos:=HPos+TMargin+fh; // вновь вычисляем отступ для дальнейшего вывода
   TextOut(LMargin,Hpos,' Текст абзаца.');
   Ellipse(LMargin*2,Hpos+LMargin,LMargin*4,Hpos+LMargin*3); // рисуем круг
   EndDoc;
 end; // конец width
end; // конец процедуры PrintAny
```

Здесь сначала были определены поля, затем по этим полям был начерчен прямоугольник, образующий рамку страницы, после чего крупным полужирным и подчеркнутым шрифтом выведено слово «Заголовок», а под ним, уже обычным и более мелким текстом – фраза «Текст абзаца». После этого был выведен круг радиусом, равным ширине левого поля. Таким образом, на один документ были введены как собственно графические объекты, так и текстовые строки, которые, впрочем, в данном случае рассматриваются тоже как графические объекты.

Пример работы этой функции можно посмотреть в программе AnyPrint, которая расположена в каталоге Demo\Part2\Print2.

Исключения и взаимодействие с API

На текущий момент мы уже знаем достаточно многое из основ ООП. Однако создание приложений под Windows в среде Delphi не ограничивается применением объектно-ориентированного подхода. В частности, иногда возникают такие ситуации, что приходится обращаться к функциям Windows API напрямую. Кроме того, нам следует рассмотреть обработку ошибок в программах, а так же осветить вопрос некоторых глобальных объектов.

Исключения и их классы

Исключительные ситуации, или исключения (exception) могут возникать по ходу выполнения программы ввиду целого ряда причин. Они могут быть вызваны как ошибками в коде программы (например, при попытке обратиться к объекту, который не был предварительно создан), при вводе пользователем неожидаемых значений (например, строки, которая не может быть приведена к числу), при ошибках работы оборудования и т.д. Любая программа, претендующая на качественную разработку, должна уметь обрабатывать все подобные исключительные ситуации.

При возникновении подобных ошибок в программах, созданных при помощи Delphi, автоматически создается объект – Exception. Класс Exception является базовым для ряда других классов исключений – EMathError, EInvalidOp, EZeroDivide и т.д. (названия всех классов, относящиеся к исключениям, принято начинать не с буквы T, а с буквы E). Он происходит непосредственно от класса TObject и имеет 2 свойства – Message и HelpContext, а так же 8 методов.

Свойство Message имеет тип string и содержит описание исключения. При возникновении ошибки этот текст используется в окне сообщения. Ас войство HelpContext определяет индекс раздела справочного файла, содержащего информацию об ошибке. Если значение этого свойства равно нулю, то оно будет ссылаться на раздел справки родительского объекта.

Что касается методов, то все они представлены различными вариантами метода Create. Сам метод Create для исключений определен следующим образом:

```
constructor Create(const Msg: string);
```

Т.е., фактически, создавая исключение, следует сразу же назначить значение его свойству Message при помощи аргумента конструктора. Другой вариант конструктора, CreateHelp, позволяет параллельно назначить значение и для второго свойства:

```
constructor CreateHelp(const Msg: string; AHelpContext: Integer);
```

Если в тексте сообщения следует привести какие-либо динамически получаемые данные, то используют вариант конструктора с суффиксом Fmt:

```
constructor CreateFmt(const Msg: string; const Args: array of const);
constructor CreateFmtHelp (const Msg: string; const Args: array of const;
AHelpContext: Integer);
```

При этом значения, указанные в массиве Args, будут подставлены в строку. Для этого используется функция Format, которой передаются строка и массив в качестве аргу-

ментов. Эта функция выполняет подстановку значений массива в места строки, выделенные при помощи символа %. Например, если строка выглядит как «Ошибка в функции %s», а массив определен как «['MyFunc']», то результатом выполнения этой функции будет «Ошибка в функции MyFunc». Соответственно, создание подобного исключения будет выглядеть следующим образом:

```
constructor CreateFmt('Ошибка в функции %s', ['MyFunc']);
```

Как уже было отмечено, класс Exception имеет ряд потомков, каждый из которых предназначен для обработки того или иного типа ошибок. Например, для математических ошибок определен класс EMathError. Однако этот класс сам по себе не используется, зато его потомки, среди которых отметим классы EInvalidOp, EOverflow, EZeroDivide, используются для оповещения о таких ситуациях, как неверная операция, переполнение буфера и попытка деления на 0, соответственно.

При возникновении исключительной ситуации создается исключение того или иного вида, на основании чего можно определить, в чем именно кроется проблема.

Вызвать исключение в программе можно и искусственным методом – при помощи ключевого слова raise. Например программа может проверять какой-либо ввод пользователя, и в том случае, если он оказывается не тем, что ожидалось, генерировать исключительную ситуацию:

```
if password <> 'password' then raise Exception.Create('Неверный пароль!');
```

Выполнение оператора, указанного после raise, приводит к возникновению исключительной ситуации. После этого дальнейшее выполнение кода процедуры прерывается, равно как и кода, вызвавшего эту процедуру, если вызов был произведен из другой подпрограммы. Перемещение исключения можно рассматривать с точки зрения всплытия, т.е. с места своего возникновения ошибка последовательно «всплывает» сначала к вызвавшей данную процедуру или функцию подпрограмме, от нее – к следующей и т.д., пока не дойдет до уровня выполнения программы, т.е. до глобального объекта Application. На этом, конечном этапе и будет выдано сообщение об ошибке.

ПРИМЕЧАНИЕ

> С некоторыми глобальными объектами, в том числе с Application, мы ознакомимся несколько позже в этой же главе.

Если при этом ошибка возникла в основном коде программы (т.е. вызвавший ошибку код был написан в самом файле проекта dpr), то на этом выполнение программы прекратится, о чем будет выдано сообщение (рис. 10.1).

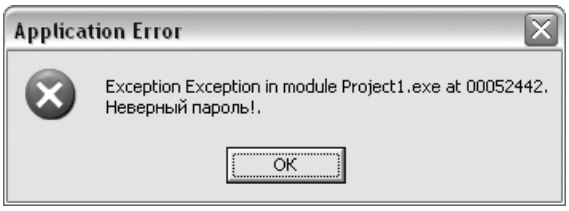

Рис. 10.1. Ошибка приложения

В том же случае, если исключительная ситуация произошла в каком-либо модуле, то программа продолжит свою работу, ожидая дальнейших действий пользователя. Од-

нако некоторые данные при этом могут оказаться утерянными (например, функция не вернет значения), или же может оказаться невыполненным какой-либо иной важный код, скажем, создающий глобальные объекты, сохраняющий информацию и т.д. Все это говорит о том, что исключительные ситуации следует обрабатывать.

Обработка исключений

Для обработки исключительных ситуаций в Delphi используются специальные операторы – try…except и try…finally. Эти операторы являются своего рода ловушками для исключительных ситуаций и позволяют разработчику приложения предусмотреть код, обрабатывающий возникшие исключения. Тем самым можно на любом этапе перехватить дальнейшее всплытие ошибки.

При помощи оператора try…except выполняет перехват ошибки, как правило, с целью ее подавления. Он имеет следующий синтаксис:

```
try
 <потенциально вызывающий исключения код>
except
 [ on <Класс исключения> do <оператор>; ]
end;
```

В том случае, если между except и end не писать никакого кода, то исключительная ситуация будет просто подавлена. Однако такое подавление чаще всего не является достаточным условием, поскольку оно не несет никакой информации ни пользователю, ни самой программе. Например, если так подавить ошибку с неверным паролем (а из-за подавления никакого сообщения выдано не будет), то пользователь такой программы может лишь догадываться, почему после того, как он сообщил пароль, ничего не происходит. В данном случае было бы правильным все-таки сообщить о том, что пароль введен не верно. Для этого используют вложенную секцию on…do:

```
try
  if password <> 'password' then raise Exception.Create('Неверный пароль!');
except
  on E: Exception do ShowMessage(E.Message);
end;
```

На сей раз в случае возникновения исключения пользователь получит уведомление о том, что же произошло. Для этого мы создали объект E, которому автоматически присваивается значение ошибки, и использовали его для вывода информации о ней. Дальнейшее выполнение программы в данном случае будет продолжено, поскольку после окончания блока try…end исключение более не существует.

На самом деле, использование такого объекта может быть необязательным, если детальная информация об ошибке не представляется необходимой. В таком случае можно использовать следующий блок обработки исключения:

```
try
  if password <> 'password' then raise Exception.Create('Неверный пароль!');
except
  on Exception do ShowMessage('ОШИБКА!');
end;
```

Что касается блоков обработки, то их может быть несколько, каждый – для своего класса исключения:

```
try
 a:=b*c/d;
except
  on EZeroDivide do ShowMessage('Делить на 0 нельзя');
  on EOverflow do ShowMessage('Слишком большое число');
  on EMathError do ShowMessage('Математическая ошибка');;
end;
```

Здесь мы определили 3 блока, и в случае возникновения той или иной исключительной ситуации, будет выдано то или иное сообщение. Этим данная часть оператора напоминает оператор case, для которого, как мы помним, существовал вариант «для остальных случаев» – else. Имеется такая возможность и здесь:

```
try
 a:=b*c/d;
except
  on EZeroDivide do ShowMessage('Делить на 0 нельзя');
  on EOverflow do ShowMessage('Слишком большое число');
  on EMathError do ShowMessage('Математическая ошибка');
 else
  ShowMessage('Общая ошибка');
end;
```

Наконец, если тип ошибки не имеет никакого значения, то можно оставить только общий обработчик, для чего не требуется даже ключевого слова else:

```
try
 a:=b*c/d;
except
 ShowMessage('Общая ошибка');
end;
```

Важно лишь отметить, что все эти блоки выполняются только тогда, когда возникает исключительная ситуация. При этом, если после ключевого слова try расположено несколько операторов, и исключение возникает в одном из них, то все последующие выполнены не будут. Вместе с тем, случаются ситуации, когда имеется код, который следует выполнить в любом случае, без оглядки на то, что случится перед этим. В таких случаях используют другой оператор – try…finally, и требующий обязательного выполнения код помещают в часть после finally. Типичным примером использования такой конструкции можно считать уничтожение объектов или иные операции освобождения памяти, а так же закрытия файлов и т.д. Например, при работе с файлами всегда следует использовать try…finally для закрытия файла:

```
try
  Rewrite(F);
  writeln(F,s);
finally
  CloseFile(F);
end;
```

В данном случае, если даже произойдет ошибка, связанная с доступом к файлу – т.е. если его не удастся открыть (например, если диск защищен от записи), или же записать в него информацию (нет места на диске), закрыт он будет в любом случае, что предотвратит возможные дальнейшие ошибки. При этом само исключение подавлено не будет, т.е. сообщение об ошибке будет выведено и дальнейшее выполнение подпрограммы (но уже после блока finally…end) будет прервано.

Но оба подхода можно комбинировать. Например, в данном случае блок try…finally можно вложить в блок try…except:

```
try
  AssignFile(F);
  try
    Rewrite(F);
    writeln(F,s);
  finally
    CloseFile(F);
  end;
except
  on E: Exception do ShowMessage(E.Message);
end;
```

Кроме этого, в Delphi допускается вкладывать однотипные обработчики ошибок друг в друга, например, один блок try…except может быть вложен в другой.

Глобальные объекты

При создании Windows-приложений нередко возникает необходимость в управлении программой в целом как отдельным объектом. Для этих целей в Delphi предусмотрен специальный объект – Application класса TApplication, представляющий программу в целом. Его использование можно увидеть в любом файле проекта VCL-приложения. Чтобы увидеть это, достаточно создать новое приложение и открыть файл проекта, для откытия которого можно воспользоваться списком модулей, вызываемого кнопкой View Unit (можно так же через главное меню – View → Units, или при помощи сочетания горячих клавиш Ctrl+F12). По умолчанию он имеет название Project1, и его стандартный код имеет вид, приведенный в листинге 10.1.

Листинг 10.1. Заготовка кода для VCL-приложения

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Уже по приведенному в листинге коду мы можем познакомиться с 3 основными методами этого объекта – Initialize, CreateForm и Run. Первый производит подготови-

тельную работу, т.е. фактически, создает объект приложения. Метод CreateForm используется для создания окон приложения, а метод Run производит фактический запуск программы на выполнение. Среди других методов приложения моно отметить такие, как Minimize и Restore, служащие, соответственно, для сворачивания программы на панель задач и для ее восстановления, а так же метод BringToFront, который выводит окно на верхнюю поверхность рабочего стола. Метод Terminate используется для прекращения работы программы (он вызывается автоматически, когда закрывается главное окно приложения). Еще 4 метода – HelpCommand, HelpContext, HelpJump и HelpKeyword – предназначены для работы со справочными файлами.

При работе приложения, в случае обработки больших массивов данных, возникают случаи, когда программа не только не реагирует на действия пользователя, но даже не может выполнить обновление собственного окна. Для того, чтобы предотвратить подобные ситуации, используют специальный метод – ProcessMessages, который предписывает приложению обработать накопившуюся очередь сообщений.

Среди свойств приложения, прежде всего, следует отметить такие, как Title, Icon и HelpFile. Свойство Title определяет заголовок программы, т.е. то, что вы видите на панели задач. Свойство Icon определяет значок («иконку») программы. Ну а свойство HelpFile связывает приложение с файлом справочной информации. Все эти свойства можно определить как программно, написав соответствующий код, так и при помощи окна свойств проекта (Project → Options), на закладке Application (рис. 10.2).

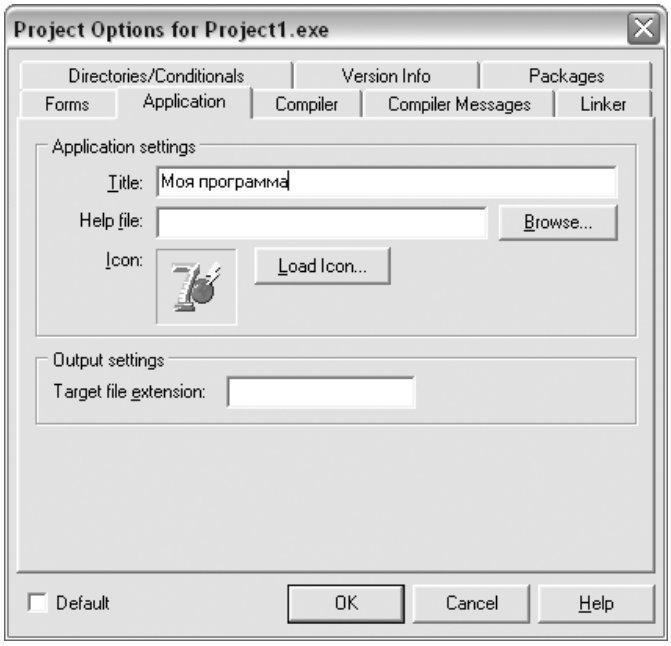

Рис. 10.2. Установка параметров приложения в окне свойств проекта

Если установить в диалоге Project Options новые значения и нажать на кнопку OK, то внесенные изменения для свойств Title и HelpFile отобразятся в коде программы. Что касается значка программы, то он хранится в отдельном файле ресурсов (res), кото-

рый присоединяется к приложению в процессе компиляции, для чего используется директива «{$R *.res}».

Поскольку любой визуальный компонент может отображать всплывающую текстовую подсказку, то для объекта Application предусмотрен ряд свойств, управляющих видом и выводом таких подсказок. В частности, цвет определяют при помощи свойства HintColor, задержку перед появлением после наведения на компонент мышки – при помощи HintPause, а время его отображения – свойством HintHidePause.

Некоторые свойства приложения доступны только во время выполнения. Среди них можно выделить свойство ExeName, содержащее информацию об имени самого исполняемого файла, включая полный путь к нему.

Помимо Application, при запуске приложения создается еще один глобальный объект, представляющий экранную среду – Screen. При помощи этого объекта можно получить информацию о разрешение экрана, установить вид курсора мыши для приложения, или узнать количество его окон. Основные свойства класса TScreen приведены в таблице 10.1.

Таблица 10.1. Основные свойства TScreen

Свойство	Тип	Описание
ActiveControl	TWinControl	Указывает, какой элемент управления в данный момент имеет фокус ввода
ActiveForm	TForm	Указывает, какое окно активно в данный момент
Cursor	TCursor	Определяет вид указателя курсора мышки для приложения
Cursors	array of HCursor	Список всех курсоров, доступных для приложения
Fonts	TStrings	Список названий всех шрифтов, доступных для вывода на экран
FormCount	Integer	Указывает на число окон (форм), созданных приложением
Forms	array of TForm	Список всех окон, созданных приложением
Height	Integer	Указывает на вертикальное разрешение экрана
HintFont	TFont	Определяет шрифт для всплывающих подсказок
IconFont	TFont	Определяет шрифт для подписей к значкам в диалогах выбора файлов
MenuFont	TFont	Определяет шрифт для меню
Width	Integer	Указывает на горизонтальное разрешение экрана
WorkAreaHeight	Integer	Указывает на высоту рабочего стола Windows
WorkAreaLeft	Integer	Указывает на координаты левого угла рабочего стола
WorkAreaRect	Integer	Указывает на координаты прямоугольника, образующего рабочий стол
WorkAreaTop	Integer	Указывает на координаты верхнего угла рабочего стола
WorkAreaWidth	Integer	Указывает на ширину рабочего стола

Использовать объекты Screen и Application можно как в главном модуле программы (файле проекта), так и в модулях отдельных форм. При использовании в главном мо-

дуле обычно устанавливают глобальные параметры, например, вид всплывающих подсказок. В частности, можно определить довольно-таки экзотический вид всплывающих подсказок, дополнив программу следующими строками:

```
Screen.HintFont.Color:=$00408080; // цвет шрифта
Screen.HintFont.Size:=14; // размер шрифта
Application.HintColor:=$0080FF80; // цвет фона
Application.HintPause:=1000; // задержка перед появлением 1 секунда
Application.HintHidePause:=2000; // время показа 2 секунды
```

Если вставить этот код в dpr-файл перед обращением к методу Application.Run, то можно будет убедиться, что через секунду после наведения курсора на окно запущенного приложения будет появляться всплывающая подсказка с крупным коричневым текстом на зеленом фоне. Разумеется, при этом для окна приложения следует установить значения свойства ShowHint в true, и написать какой-либо текст для свойства Hint. Впрочем, это можно сделать не только через инспектор объекта в процессе разработки приложения, но и программно, поместив соответствующий код после создания формы. В результате мы получим код, приведенный в листинге 10.2.

Листинг 10.2. Использование объектов Application и Screen

```
program app_scr;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Form1.Hint:='Ну и подсказочка!';
  Form1.ShowHint:=true;
  Screen.HintFont.Color:=$00408080;
  Screen.HintFont.Size:=14;
  Application.HintColor:=$0080FF80;
  Application.HintPause:=1000;
  Application.HintHidePause:=2000;
  Application.Run;
end.
```

Здесь же можно установить и такие параметры, как заголовок программы, используя свойство Title объекта Application:

```
Application.Title:='Super Hint!';
```

Кроме того, можно поэкспериментировать с такими свойствами объекта Screen, как Height и WorkAreaHeight, причем для вывода информации можно использовать заголовок главного окна:

```
Form1.Caption:='Экран '+IntToStr(Screen.Height)+', рабочий стол '+
IntToStr(Screen.WorkAreaHeight);
```

В данном случае в строку Uses потребуется дописать модуль SysUtils, поскольку использованная здесь функция IntToStr расположена именно в этом модуле. Окончательный вариант программы можно найти в каталоге Demo\Part2\Global.

Работа с INI-файлами

При разработке приложений часто встает вопрос о том, где и как хранить информацию, связанную с его настройками. Нередко для этих целей используются специальные INI-файлы, которые хранят в себе информацию, разбитую по логическим группам в виде «ключ-значение». В Delphi имеется класс, обеспечивающий простую работу с такими файлами – TIniFile. Чтобы приложение могло получить доступ к этому классу, в секцию используемых модулей следует добавить inifiles.

Имя файла, ассоциированного с объектом типа TIniFile, задается непосредственно при создании экземпляра этого класса, в конструкторе Create:

```
var MyIni: TIniFile;
...
TIniFile.Create('myfile.ini');
```

Впоследствии можно узнать, какой файл ассоциирован с данным объектом при помощи его свойства FileName, однако изменить его уже не получится. Вместе с тем, у TIniFile имеется свыше 20 методов, при помощи которых можно считывать, проверять и изменять содержимое INI-файла. Все они приведены в таблице 10.2.

Таблица 10.2. Методы класса TIniFile

Метод	Принимаемые параметры	Описание
DeleteKey	const Section, Ident: String	Удаляет указанный ключ из INI файла
EraseSection	const Section: String	Удаляет содержимое указанной секции в INI файле
ReadSection	const Section: String; Strings: TStrings	Считывает имена всех ключей в указанной секции и заносит их в список строк
ReadSections	Strings: TStrings	Считывает названия всех секций в файле и заносит их в список строк
ReadSectionValues	const Section: String; Strings: TStrings	Считывает все значения в указанной секции и заносит их в список строк
ReadString	const Section, Ident, Default: String	Считывает и возвращает значение-строку из указанного ключа
WriteString	const Section, Ident, Value: String	Записывает значение-строку в указанный ключ
ReadBool	const Section, Ident: String; Default: Boolean	Считывает и возвращает булево значение из указанного ключа
ReadDate	const Section, Ident: String; Default: TDateTime	Считывает и возвращает значение-дату из указанного ключа
ReadDateTime	const Section, Ident: String; Default: TDateTime	Считывает и возвращает значение-дату и время из указанного ключа
ReadFloat	const Section, Ident: String; Default: Double	Считывает и возвращает значение-вещественное число из указанного ключа
ReadInteger	const Section, Ident: String; Default: Longint	Считывает и возвращает значение-целое число из указанного ключа
ReadTime	const Section, Ident: String; Default: TDateTime	Считывает и возвращает значение-время из указанного ключа
SectionExists	const Section: String	Проверяет INI файл на наличие указанной

		секции
WriteBool	const Section, Ident: String; Value: Boolean	Записывает булево значение в указанный ключ
WriteDate	const Section, Ident: String; Value: TDateTime	Записывает значение-дату в указанный ключ
WriteDateTime	const Section, Ident: String; Value: TDateTime	Записывает значение-дату и время в указанный ключ
WriteFloat	const Section, Ident: String; Value: Double	Записывает значение-вещественное число в указанный ключ
WriteInteger	const Section, Ident: String; Value: Longint	Записывает значение-целое в указанный ключ
WriteTime	const Section, Ident: String; Value: TDateTime	Записывает значение-время в указанный ключ
ValueExists	const Section, Ident: String	Проверяет INI файл на наличие указанного ключа в определенной секции

Таким образом, можно без каких-либо дополнительных накладных расходов (с точки зрения написания собственного кода), создавать и считывать стандартные INI-файлы. Например, мы можем создать приложение, которое сможет «запоминать» введенную информацию и отображать ее при следующем запуске. В принципе, мы уже делали нечто подобное еще при создании программы «угадывания чисел», рассмотренной в первой части. Однако тогда мы лишь последовательно записывали в файл пару строк, а затем таким же образом их считывали. Но если бы нам требовалось сохранить большее количество значений, то мы столкнулись бы с трудностями такого рода, как невозможность идентифицировать то или иное значение при просмотре файла. Кроме того, пришлось бы постоянно держать в уме, какая по строка что должна хранить. Использование INI-файлов решает эту задачу.

Для примера возьмем консольное приложение, которое будет последовательно спрашивать различную информацию у пользователя, а затем сохранит ее в указанном файле. При следующем запуске она сможет считать этот файл и вывести информацию из него на экран. Основное тело программы при этом может получиться примерно таким, как показано в листинге 10.3.

Листинг 10.3. Название листинга

```
program myini;
{$APPTYPE CONSOLE}
uses
  SysUtils, IniFiles;
var
 ans: Char;
 fn: string;
begin
 write('Load data from an INI file? [Y/N]');
 readln(ans);
 if (ans='Y') or (ans='y') then begin
   write('Please input file name: ');
   readln(fn);
```

```
   fn:='c:\'+fn+'.ini';
   if FileExists(fn) then begin
     ShowData(fn);
   end else begin
     writeln('File not found and will be created.');
     FillData(fn);
   end;
 end else begin
   write('Please input file name to save data: ');
   readln(fn);
   fn:='c:\'+fn+'.ini';
   FillData(fn);
 end;
 readln(fn);
end.
```

Прежде всего, наша программа интересуется, хочет ли пользователь просмотреть информацию из уже существующего файла, или нет, и если хочет, то запрашивает имя файла. Здесь мы подразумеваем, что пользователь будет вводить только имя файла, без пути и расширения, которые добавляются автоматически. Затем стандартная функция FileExists проверяет получившийся файл на существование, после чего либо выводит его содержимое при помощи процедуры ShowDate (которую нам еще предстоит создать), либо выводит сообщение о том, что файл не найден, но будет создан. После этого программа обращается к процедуре FillData, которая так же будет нами написана для ввода информации и сохранения ее в INI-файле. Эта же функция будет вызвана и в том случае, если пользователь изначально откажется от вывода информации, в таком случае программа предварительно запросит имя файла для дальнейшего сохранения.

Теперь, когда основа программы готова, можно определиться, какие данные мы хотим хранить, и какой для этого понадобится формат файла. Допустим, мы хотим сохранить информацию 2-х категорий: персональную и рабочую. В таком случае наш INI файл будет состоять из 2 секций, скажем, Userdata и Jobdata. В первой секции сохраним имя (Name) и возраст (Age), а во второй – должность (Title) и оклад (Salary). Процедура, отвечающая за вывод информации, получится достаточно простой – в ней достаточно создать INI-файл с указанным именем и последовательно считывать информацию, попутно выводя ее на экран. Например, для строкового значения мы получим следующий код:

```
writeln('Name..... '+IniF.ReadString('Userdata','Name','Anonymous'));
```

Если же речь идет о числовом значении, то нам придется предварительно преобразовать его в строку:

```
writeln('Age..... '+IntToStr(IniF.ReadInteger('Userdata','Age',0)));
```

Несколько сложнее получится код процедуры для записи файла, что, впрочем, связано не с самой записью данных, а в том, что их предварительно следует получить от пользователя. Поэтому там, где мы при выводе обходились одной строкой кода, для ввода понадобится целых 3, а так же переменная для хранения вводимого значения:

```
write('Name: ');
readln(s);
```

```
IniF.WriteString('Userdata','Name',s);
```

Подобный код потребуется выполнить для каждого поля данных, при этом нам понадобятся 3 различных переменных для хранения данных 3 типов (дважды – строк, и по разу целое и вещественное числа). Предварительно следует не забыть создать переменную типа TIniFile, и вывести пояснительный текст, а к завершению работы процедуры освободить память, занимаемую более не нужной переменной. Последнее условие следует выполнить и в процедуре ShowData. В итоге мы получим код, приведенный в листинге 10.4.

Листинг 10.4. Процедуры сохранения и считывания INI-файлов

```
procedure FillData(fn: string);
var
 IniF: TIniFile;
 s: string;
 i: integer;
 f: double;
begin
 IniF:=TIniFile.Create(fn);
 writeln('Please fill a form...');
 write('Name: ');
 readln(s);
 IniF.WriteString('Userdata','Name',s);
 write('Age: ');
 readln(i);
 IniF.WriteInteger('Userdata','Age',i);
 write('Position: ');
 readln(s);
 IniF.WriteString('Jobdata','Title',s);
 write('Salary: ');
 readln(f);
 IniF.WriteFloat('Jobdata','Salary',f);
 IniF.Free;
end;
procedure ShowData(fn: string);
var
 IniF: TIniFile;
begin
 IniF:=TIniFile.Create(fn);
 writeln('Name..... '+IniF.ReadString('Userdata','Name','Anonymous'));
 writeln('Age..... '+IntToStr(IniF.ReadInteger('Userdata','Age',0)));
 writeln('Position.. '+IniF.ReadString('Jobdata','Title','Unemployed'));
 writeln('Salary....
'+FloatToStrF(IniF.ReadFloat('Jobdata','Salary',0.00),ffFixed,6,2));
 IniF.Free;
end;
```

С полным исходным кодом программы можно ознакомиться в примере, расположенном в каталоге Demo\Part2\IniFiles.

Работа с реестром Windows

Файлы INI и класс TIniFiles – достаточно удобный способ хранения различной настроечной информации. Тем не менее, начиная с Windows 95, появилось централизованное хранилище для настроек системы и всех установленных программ – реестр (Registry). При разработке приложений в Delphi удобнее всего работать с реестром, используя класс TRegistry. Чтобы включить объявление этого класса, следует указать модуль registry в списке uses.

Реестр Windows имеет несколько ключевых разделов, в чем можно убедиться, открыв имеющуюся в Windows программу редактирования реестра (regedit). В частности это разделы HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_USERS, HKEY_LOCAL_MACHINE и HKEY_CURRENT_CONFIG. Чтобы приступить к работе с реестром из программы, требуется указать один из разделов. Делается это при помощи свойства RootKey:

```
var Reg: TRegistry;
...
Reg:=TRegistry.Create;
Reg.RootKey:=HKEY_CURRENT_USER;
```

Далее в ход идут методы класса TRegistry. В частности, за выбор раздела реестра, из которого надо будет считывать данные, используется метод OpenKeyReadOnly. В качестве аргумента ему передается адрес раздела реестра, например:

```
Reg.OpenKeyReadOnly('\SOFTWARE\MySoft\TestApp');
```

Если указанный раздел существует, и к нему может быть обеспечен доступ, то обращение к данному методу вернет истину. Если же раздела может не существовать, или если требуется открыть раздел на запись, то используют метод OpenKey:

```
Reg.OpenKey('\SOFTWARE\MySoft\TestApp',true);
```

Для него в качестве 2-го параметра указывают булево значение, которое указывает на то, должен ли указанный раздел быть создан, если его не существует. В результате выполнения приведенного кода раздел, при необходимости, будет создан и открыть на чтение и запись. Если же требуется только создать новый раздел, то используют метод CreateKey:

```
Reg.CreateKey('\SOFTWARE\MySoft\TestApp');
```

Для удаления раздела используют метод DeleteKey, а для проверки указанного раздела на существование – KeyExists. Подобно методу CreateKey, эти методы так же принимают адрес раздела и возвращают ложь или истину в зависимости от результата операции.

Если же требуется выполнить проверку на наличие значения в текущем открытом разделе, то используют метод ValueExists, которому в качестве аргумента передают имя значения.

Что касается записи и считывания значений, то, подобно классу TIniFile, для TRegistry определен ряд методов для взаимодействия с данными различных типов, причем для реестра к типам Boolean, String, Double, Integer и даты-времени, добавляется еще и Currency. Соответственно, мы имеем 8 пар методов для этих целей.

Для примера рассмотрим приложение, состоящее из единственного окна, которое будет «запоминать» свои размеры и расположение на экране. Для этого создадим

новое VCL-приложение (File → New → Application), щелкнем сначала по его форме (Form1), а затем – по окну инспектора объекта (Object Inspector). В нем выберем закладку Events (события), найдем событие OnClose и дважды щелкнем по строке напротив. В результате мы получим заготовку для процедуры TForm1.FormClose, в которую нам надо будет добавить объявление переменной для реестра:

```
var
 Reg: TRegistry;
```

Затем в теле функции напишем следующие строки:

```
Reg:=TRegistry.Create;
Reg.RootKey:=HKEY_CURRENT_USER;
Reg.OpenKey('\SOFTWARE\MySoft\TestApp',true);
Reg.WriteInteger('left',Form1.Left);
Reg.WriteInteger('top',Form1.Top);
Reg.WriteInteger('height',Form1.Height);
Reg.WriteInteger('width',Form1.Width);
Reg.Free;
```

Вначале мы создаем экземпляр класса, затем выбираем корневой раздел, после чего открываем ключ на запись (он будет создан при необходимости), и последовательно заносим в него пространственные координаты окна. В завершение работы этой процедуры мы экземпляр класса удаляется из памяти за ненадобностью.

Теперь рассмотрим считывание из реестра, для чего создадим процедуру, обрабатывающую событие создания окна, для чего в инспекторе объекта найдем событие OnCreate и сделаем двойной щелчок напротив него. В получившейся процедуре нам так же понадобится сначала объявить переменную Reg, затем создать экземпляр класса и установить корневой раздел. Затем следует открыть раздел на чтение, причем если это окажется невозможным (а при первом запуске так и будет, поскольку раздел будет создан только после выхода из программы), то считывать ничего не потребуется. Поэтому задействуем условный оператор:

```
if Reg.OpenKeyReadOnly('\SOFTWARE\MySoft\TestApp') then begin
...
end;
```

После этого остается считать все нужные данные из реестра, присваивая хранящиеся в них значения соответствующим свойствам Form1. Например, для высоты и ширины мы получим:

```
Form1.Height:=Reg.ReadInteger('height');
Form1.Width:=Reg.ReadInteger('width');
```

Вместе с тем, было бы полезным все-таки проверять наличие запрашиваемых значений в реестре, чтобы избежать возникновения исключительных ситуаций. Для этого всякий раз надо будет проверять ключ на существование:

```
if Reg.ValueExists('width') then Form1.Width:=Reg.ReadInteger('width');
```

В результате код этого модуля программы получит приблизительно такой вид, как показано в листинге 10.5.

Листинг 10.5. Сохранение координат и размеров окна в реестре

```
unit Unit1;
```

```
interface
uses
  Windows, Forms, Registry;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
procedure TForm1.FormCreate(Sender: TObject);
var
 Reg: TRegistry;
begin
 Reg:=TRegistry.Create;
 Reg.RootKey:=HKEY_CURRENT_USER;
 if Reg.OpenKeyReadOnly('\SOFTWARE\MySoft\TestApp') then begin
   if Reg.ValueExists('left') then
     Form1.Left:=Reg.ReadInteger('left');
   if Reg.ValueExists('top') then
     Form1.Top:=Reg.ReadInteger('top');
   if Reg.ValueExists('height') then
     Form1.Height:=Reg.ReadInteger('height');
   if Reg.ValueExists('width') then
     Form1.Width:=Reg.ReadInteger('width');
 end;
 Reg.Free;
end;
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
 Reg: TRegistry;
begin
 Reg:=TRegistry.Create;
 Reg.RootKey:=HKEY_CURRENT_USER;
 Reg.OpenKey('\SOFTWARE\MySoft\TestApp',true);
 Reg.WriteInteger('left',Form1.Left);
 Reg.WriteInteger('top',Form1.Top);
 Reg.WriteInteger('height',Form1.Height);
 Reg.WriteInteger('width',Form1.Width);
 Reg.Free;
end;
end.
```

С исходным кодом приложения так же можно ознакомится, посмотрев его в каталоге Demo\Part2\Registry.

Процедуры и функции стандартных диалогов

В Delphi предусмотрено несколько процедур и функций, предназначенных для вывода простых диалоговых окон. В частности, процедура ShowMessage и функция MessageDlg позволяют вывести сообщение, а функции InputBox и InputQuery отображают окно для ввода информации.

Простейшим вариантом вывода сообщения является использование процедуры ShowMessage. Она отображает переданную ей в качестве аргумента строку на простом диалоговом окне с единственной кнопкой OK. Типичный пример использования этой процедуры – информирование пользователя о выполнении той или иной части программы:

```
ShowMessage('Формат диска C: завершен');
```

Кроме самой процедуры ShowMessage, имеются 2 других варианта – ShowMessagePos и ShowMessageFmt. Первый позволяет вывести диалоговое окно в определенном месте, что достигается путем указания координат по горизонтали и вертикали:

```
ShowMessagePos('Формат диска C: завершен',100,200);
```

Второй позволяет вывести отформатированную строку, используя обращение к функции Format, как и в случае с конструктором исключений. Таким образом, для вывода сообщения с переменной частью предпочтительно использовать именно этот вариант процедуры:

```
ShowMessageFmt('Формат диска %s завершен',['C:']);
```

Все варианты процедуры ShowMessage выводят окно с единственной кнопкой OK, при этом, разумеется, никакого значения не возвращается. В том же случае, если сообщение выводится для того, чтобы запросить у пользователя подтверждения на то или иное действие, то нам, во-первых, потребуется функция – чтобы получить вариант ответа, а так же возможность указать возможные варианты. Все это мы имеем в лице функции MessageDlg, которая имеет следующее определение:

```
function MessageDlg(const Msg: string; DlgType: TMsgDlgType; Buttons:
TMsgDlgButtons; HelpCtx: Longint): Word;
```

Здесь сразу же требуется прояснить 2 момента: тип диалога и тип кнопок. За тип диалога отвкечает 2-й параметр, который имеет тип TMsgDlgType и может принимать одно из следующих значений:

- mtWarning – диалог типа «предупреждение», имеет заголовок «Warning» и рисунок, изображающий восклицательный знак на фоне желтого треугольника;
- mtError – диалог типа «ошибка», имеет заголовок «Error» и изображение косого креста в красном круге;
- mtInformation – диалог типа «информация», имеет заголовок «Information» и значок со стилизованной буквой «i» в синих тонах;
- mtConfirmation – диалог типа «подтверждение», имеет заголовок «Confirm» и рисунок с зеленым вопросительным знаком;
- mtCustom – диалог произвольного типа, имеет заголовок, соответствующий имени выполняемого файла и не содержит изображения.

ПРИМЕЧАНИЕ

Внешний вид изображений, символизирующих диалог того или иного типа, периодически претерпевает некоторые изменения, в зависимости от версии Delphi.

Следующий параметр, имеющий перечисляемый тип TMsgDlgButtons, позволяет указать, какие кнопки должны быть расположены на диалоговом окне. Всего предусмотрено 11 вариантов кнопок, среди них предусмотрены такие, как OK, Cancel, Yes, No и т.д. При этом каждая такая кнопка (кроме Help), будучи нажатой пользователем, закрывает окно, а функция возвращает значение, соответствующее нажатой кнопке. Все варианты кнопок и возвращаемые ими значения, приведены в таблице 10.3.

Таблица 10.3. Варианты кнопок и значения, возвращаемые при их нажатии

Значение	Описание	Возвращаемый результат
mbYes	Кнопка с надписью «Yes» (да)	mrYes
mbNo	Кнопка с надписью «No» (нет)	mrNo
mbOK	Кнопка с надписью «OK»	mrOk
mbCancel	Кнопка с надписью «Cancel» (отмена)	mrCancel
mbAbort	Кнопка с надписью «Abort» (прервать)	mrAbort
mbRetry	Кнопка с надписью «Retry» (повторить)	mrRetry
mbIgnore	Кнопка с надписью «Ignore» (игнорировать)	meIgnore
mbAll	Кнопка с надписью «All» (все)	mrAll
mbNoToAll	Кнопка с надписью «No to All» (нет для всех)	mrNoToAll
mbYesToAll	Кнопка с надписью «Yes to All» (да для всех)	mrYesToAll
mbHelp	Кнопка с надписью «Help» (справка)	-

Следует оговориться, что все возвращаемые значения, на самом деле, являются целыми числами, что видно по определению функции. Но поскольку запомнить, что, к примеру, возвращаемое значение для OK – это 1, а для Yes – 6, весьма проблематично, то на практике вместо них используются константы, которые как раз и были приведены в таблице 10.3.

Что касается вариантов использования этой функции, то оно сводится к тому, что пользователю выводится какое-либо сообщение, предусматривающее возможность того или иного ответного действия:

```
MessageDlg('Ошибка чтения с диска. Продолжить?', mtError, [mbRetry,
mbAbort], 0);
```

Поскольку эта функция возвращает то или иное значение, то ее использование часто сопровождается условным оператором:

```
if MessageDlg('Форматировать диск C:?',mtConfirmation,[mbYes,mbNo],0) =
mrYes then FormatDriveCProc();
```

Другой вариант, для случая с множественными вариантами ответа – использование совместно с оператором-переключателем:

```
case MessageDlg('Файл изменен. Сохранить перед выходом?', mtWarning, [mbYes,
mbNo, mbCancel], 0) of
 mrYes: begin SaveFileProc(); Close; end;
 mrNo: Close;
 mrCancel: exit;
```

```
end;
```

Подобно процедуре ShowMessage, для функции MessageDlg так же предусмотрен вариант с позиционированным выводом окна. Такой вариант этой функции называется MessageDlgPos. Ее отличие от MessageDlg состоит в том, что к списку аргументов добавлено еще 2 параметра, отвечающих за расположение окна. Такой вариант используется, например, при поиске с заменой в текстовых редакторах:

```
MessageDlgPos('Заменить это вхождение?', mtConfirmation, [mbYes, mbNo], 0,
X, Y);
```

Все рассмотренные нами подпрограммы применяются для вывода сообщений. Что касается ввода, то для этих целей, как уже отмечалось, используют функции InputBox и InputQuery. Обе они выводят окно, позволяющее пользователю ввести какое-либо значение – число или строку. Различие между ними состоит лишь в том, что InputBox возвращает непосредственно результат (в виде строки), а InputQuery – истину или ложь, в зависимости от того, нажмет пользователь OK или Cancel. При этом само значение возвращается в качестве одного из параметров. В итоге мы имеем следующий синтаксис для этих функций:

```
function InputBox(const ACaption, APrompt, ADefault: string): string;
function InputQuery(const ACaption, APrompt: string; var Value: string):
Boolean;
```

Таким образом, какую из функций лучше использовать в данный момент, зависит от контекста применения. Например, если надо просто получить какое-либо значение от пользователя, то можно использовать функцию InputBox:

```
UserName := InputBox('Запрос','Введите ваше имя','анонимно');
```

В данном случае последний параметр функции будет использован в качестве значения по умолчанию (рис. 10.3).

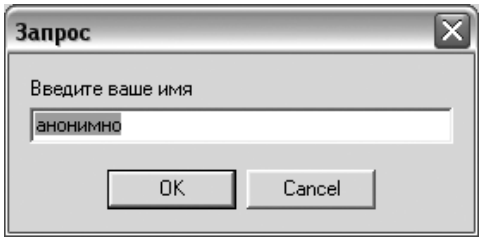

Рис. 10.3. Диалоговое окно функции InputBox

Если же в зависимости от того, введет или нет пользователь новое значение, должна быть выполнена та или иная ветвь алгоритма, то предпочтительнее использовать функцию InputQurey:

```
if InputQurey('Курс доллара ','Введите новый курс',NewCur) then UpdatePrc();
```

Помимо приведенных здесь процедур и функций, в VCL имеется ряд иных подпрограмм, использующих диалоговые окна, включая такие, как диалог выбора каталога или файла. Но поскольку их использование сопряжено с некоторыми неудобствами, в частности, им приходится передавать большое число параметров, то на практике для тех же целей чаще используют компоненты. Например, функция PromptForFileName используется для вывода диалога сохранения или открытия файла. Но более типичным (и удобным!) вариантом обращения к таким диалогам является использование

таких стандартных компонент VCL, как TOpenDialog и TSaveDialog, с которыми мы познакомимся в следующей части этой книги.

Обработка сообщений и Windows API

Как ни широк охват VCL, иногда все-таки возникает потребность в обращении к функциям Windows напрямую. Например, для того же самого вывода окна с текстовым сообщением можно использовать собственную функцию Windows API – MessageBox:

```
MessageBox(0, 'Текст сообщения', 'Заголовок', MB_OK);
```

Необходимость использования функций Windows API может быть вызвана, например, соображениями компактности исполняемого файла: использование диалогов Delphi автоматически подразумевает использование целого рада модулей, необходимых для оконного интерфейса. Если же в самой программе такие модули (например, forms) не задействуются, то их включение в исполняемый код только ради диалога не является хорошей идеей.

В то же время, обращение к функциям Windows API может быть вызвано, например, необходимостью перехвата непредусмотренных в Delphi сообщений.

ПРИМЕЧАНИЕ

> Еще одной темой, важной для дальнейшего изучения программирования в Windows вообще и в среде Delphi в частности, является концепция событийного программирования. Дело в том, что хотя ОС Windows, в отличие от Delphi, и не является объектной средой, подход к организации взаимодействия приложений (как с пользователем, так и с системой), основан на одном и том же, а именно – на событиях.

Как мы уже знаем, для событий в VCL используются обработчики событий. Но важно знать, что каждое событие порождает сообщение. Таким образом, отслеживая поступающие сообщения и отсылая собственные, мы можем действовать в обход ограничений VCL.

Для отправки сообщений чаще всего используют функции SendMessage и PostMessage. Обе они выполняют отправку сообщения конкретному окну, разница заключается лишь в том, что SendMessage ожидает ответа от получившего сообщение обработчика, а PostMessage возвращает ответ немедленно. Ценность этих функций состоит в том, что в отличие от средств, предоставляемых VCL, они могут взаимодействовать не только в рамках одного приложения, но и между совершенно разными программами и даже устройствами.

Хотя детальное ознакомления с работой Windows API явно не вписывается в рамки данной книги (не забываем, что Delphi была создана как раз для того, чтобы скрыть сложную и неуклюжую API Windows), отметим все-таки некоторые связанные с ней аспекты. Прежде всего, это касается типов данных. Хотя ранние версии Windows были написаны на Pascal, со временем Microsoft перешла на использование C и C++, поэтому типы данных в представлении Windows несколько отличаются от таковых в Delphi. Прежде всего, это касается строк: при работе с Windows напрямую следует использовать не обычные, а C-строки. В Object Pascal для этого предусмотрен специальный тип данных – PChar, а так же функции для преобразования строк одного вида

в другой. Так, для преобразования Pascal-строк в C-строки используют функцию StrPCopy, а для обратного преобразования – функцию StrPas.

```
var
 a: PChar;
 s: string;
...
s:='Строка';
new(a); // для C-строк следует предварительно выделять память
StrPCopy(a,s); // содержимое Pascal-строки s скопировано в C-строку a
s:=StrPas(a);
```

Другие типы данных, часто используемые при работе с API – целые числа, булевы значения и указатели. В таких случаях можно использовать стандартные для Object Pascal типы данных, а к нужному виду они, при необходимости, будут приводиться автоматически.

СОВЕТ

> В поставку Delphi включена документация по Windows API. Ссылки на файлы вы найдете в разделе MS SDK Help Files, вложенном в раздел Help программной группы Delphi в меню кнопки пуск. Наибольший интерес с точки зрения изучения функций API представляет собой файл Win32 Programmer's reference.

Что касается VCL, то в Delphi все же имеется специальный компонент, который может отлавливать все сообщения, адресуемые приложению. Для этого существует компонент AppEvents, который принимает все сообщения, адресованные объекту Application. Среди событий, отслеживаемых компонентом AppEvents, выделим OnMessage – именно это событие происходит, когда приложение получает сообщение от Windows или иной программы. Кроме того, ряд компонент, на самом деле, являются оболочкой для вызова тех или иных функций Windows. Впрочем, о компонентах Delphi будет рассказано в следующей части этой книги.

Часть III.
Визуальное программирование

Работа с VCL в среде Delphi

Изучив основы программирования в среде Delphi, включая даже такую область, как объектно-ориентированный подход, мы подошли к главной цели – вопросу быстрого создания приложений при помощи Delphi IDE и предоставляемых VCL компонент. Но прежде, чем исследовать сами компоненты, разберемся с вопросом их добавления и редактирования, а так же познакомимся поближе с фундаментом для размещения всех остальных компонент – классом TForm.

Работа с визуальным редактором

Когда вы только создаете новый проект в Delphi, вы сразу же получаете не только файл проекта, но и готовое к дальнейшему использованию окно программы. В терминологии Delphi это окно называется формой (form) и представляет собой основу, на которую помещаются все остальные компоненты приложения. Эта форма будет называться Form1, поскольку автоматическое наименование компонент в Delphi производится по принципу «название компонента + порядковый номер». Таким образом, для первого по счету компоненту типа Form мы получим название Form1. Кроме того, для компонент, имеющих текстовые подписи, включая ту же форму, это же название используется для свойства типа Caption или Text.

Чтобы поместить новый элемент интерфейса на поверхность формы, следует выбрать нужный компонент из палитры, щелкнув по нему мышкой, а затем щелкнуть по тому месту на форме, где этот компонент должен находиться по вашему замыслу. Например, если щелкнуть по компоненте Button (кнопка), находящейся на закладке Standard, а затем – по центру формы, то на форме появится кнопка, причем ее левый верхний угол будет, по возможности, находиться как раз в том месте, где был произведен щелчок мышкой (рис. 11.1).

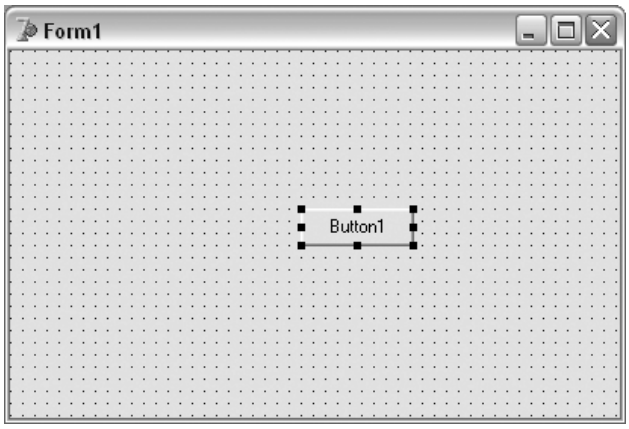

Рис. 11.1. Форма с помещенной на нее кнопкой

При этом появившийся на форме объект будет заключен в «рамку», обозначенную 8 черными квадратиками по углам и по серединным точкам сторон. Наличие такой рамки на объекте означает, что он в данный момент является выбранным для редактирования, и именно его свойства отображаются в данный момент в окне инспектора объектов.

Впрочем, эта рамка имеет еще одно предназначение: так, если потянуть за квадратик мышкой, то размер компонента будет изменен путем растягивания или сжатия в выбранном направлении. Переместить же компонент, можно просто перетащив его мышкой, подобно тому, как это делается с ярлыками на рабочем столе Windows. При этом «сетка», к которой привязываются компоненты, имеет размер 8 на 8 пикселей. Впрочем, при желании сетку можно перенастроить, или отключить вообще, для чего следует из меню Tools выбрать пункт Environment Options и установить нужные значения в группе Grid Options, находящейся на закладке Designer.

Но вернемся к тому, что можно сделать с компонентой при помощи инспектора объекта. Разумеется, учитывая то, что расположение компоненты на форме – не более, чем визуальное отображение свойств Left и Top, а ее размеры – свойств Height и Width, то, внося изменения в соответствующие поля инспектора объекта, можно добиться максимально точного расположения.

Вместе с тем, если компонент необходимо, скажем, отцентрировать по отношению к форме, или, что еще более актуально – выровнять интервалы между группой компонент, то можно воспользоваться утилитой выравнивания. Для этого следует щелкнуть по компоненте правой клавишей мышки и из контекстного меню выбрать пункт Position → Align, в результате чего откроется окно Alignment (рис. 11.2).

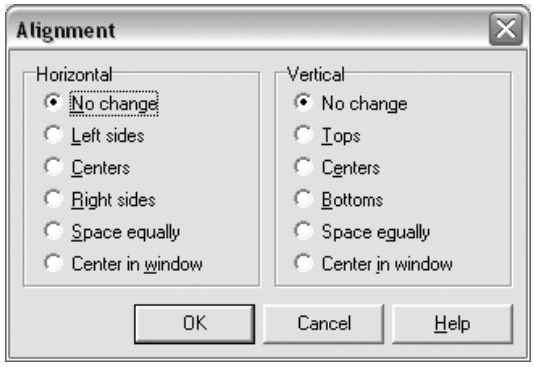

Рис. 11.2. Выравнивание компонент

Выравнивать можно как по горизонтали, так и по вертикали. Допустим, если у нас имеется единственная кнопка, которую мы хотим поместить в центр окна, то следует выбрать пункты Center in window как для горизонтального (Horizontal), так и для вертикального (Vertical) выравнивания.

Теперь рассмотрим вариант, когда следует выровнять сразу несколько объектов. Для этого поместим на форму еще 2 кнопки, в произвольном месте. Затем следует выделить их. Сделать это можно двумя способами. В том случае, если форма пустая, а область, занимаемая нужными компонентами, не содержит других компонент (т.е. как в нашем случае), то достаточно охватить их общей рамкой при помощи мышки,

нажав ее левую кнопку в одном углу группы (но не на области компонента), и протянув виртуальный прямоугольник до противоположного угла. Таким образом, все компоненты, попавшие хотя бы одним краем в такой прямоугольник, окажутся выделенными, что будет отмечено 4 серыми квадратиками по углам каждого компонента (рис. 11.3). Альтернативным способом является выбор компонент путем последовательного щелканья по ним мышкой при удерживаемой клавише Shift.

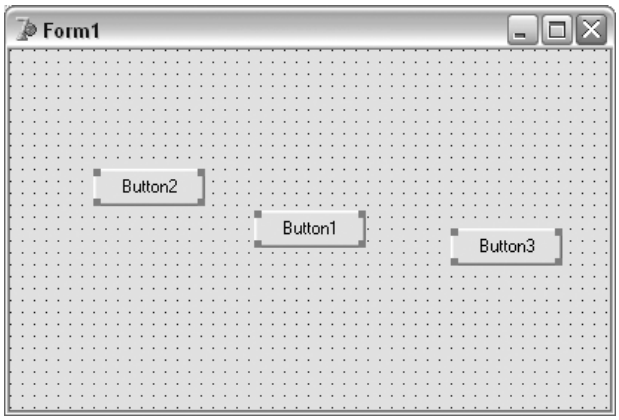

Рис. 11.3. Выбранная группа компонент

Когда выбрано несколько компонент, в инспекторе объектов отображаются только те свойства, которые можно изменить коллективно, для всех выбранных компонент сразу. Но если дело касается центрирования или взаимного выравнивания компонент, то лучше, опять-таки, обратиться к окну выравнивания. Так, чтобы выровнять все кнопки по центру формы по вертикали, а по горизонтали их сделать равноудаленными друг от друга, в окне выравнивания следует отметить Space equally в Horizontal и Center in window – в Vertical. В результате кнопки выстроятся в стройный ряд по середине окна (рис. 11.4).

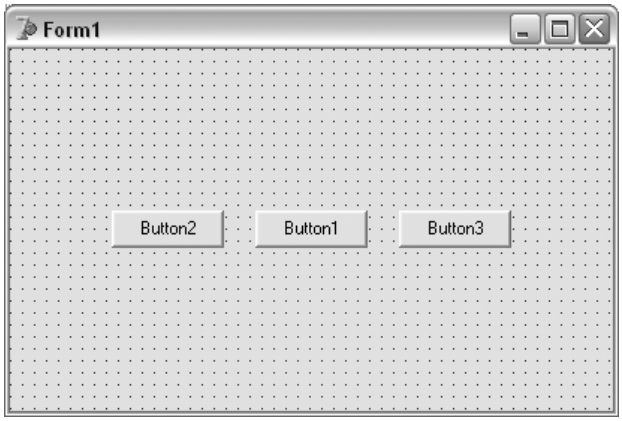

Рис. 11.4. Группа после выравнивания

С помощью другого инструмента – Size (размер) можно упорядочить размеры у группы элементов. Для вызова диалогового окна Size следует выбрать из контекстно-

го меню пункт Position → Size, после чего можно будет изменять размеры по вертикали и горизонтали, приводя их к наибольшему или наименьшему общему, или же указывая размеры явно, в пикселях. Прочем, последнее можно сделать и из инспектора объектов, задав нужные значения для свойств Height и Width.

В остальном, правка свойств как отдельных компонент, так и их групп, производится путем установки соответствующих значений в окне инспектора объектов. Если при этом свойство отмечено значком «+», то это говорит о том, что оно составное, и для установки всех параметров следует сначала щелкнуть по этому значку, а затем выбрать значения во всех раскрывшихся полях. Что касается допустимых значений полей, то они так же определяются описанием класса данного компонента. При этом инспектор объекта, как правило, просто не допускает установки заведомо недопустимых значений. Так, для свойств булевского типа можно выбрать только True или False, а для полей перечисляемого типа будет приведен список с допустимыми значениями. Кроме того, для тех свойств, которые задаются, скажем, целыми числами, значениями, вы не сможете указать строку или вещественное число. Все это автоматически исключает целый ряд возможных ошибок.

СОВЕТ

> Еще раз напомню, что нажатие на клавишу F1 в момент, когда курсор ввода находится на том или ином поле в инспекторе объектов, приводит к вызову справочной информации по данному свойству объекта.

Наконец, открою один маленький секрет: если произвести двойной щелчок по любому полю-списку, то будет выбрано следующее значение. Но если дважды щелкнуть по списку свойства Color, то откроется стандартное для Windows окно выбора цвета. Вместе с тем следует отметить, что такое поведение – открытие дополнительного окна редактирования – типично для тех свойств, которые имеют кнопку с многоточием, например, Font или Hint.

Формы в Delphi

Приложения, создаваемые в среде Delphi, как, впрочем, и любые другие Windows-программы, сосредоточены вокруг форм – окон приложения. К каждой форме в Delphi привязываются 2 файла. Один из них, с типичным для языка Pascal расширением pas, является программным модулем, описывающим класс формы и все ее компоненты. Другой файл, имеющий тип dfm, описывает расположение всех компонент на форме, их свойства, а так же параметры самой формы – т.е. все то, что можно настроить при помощи инспектора объектов. Это сделано, в том числе, с той целью, чтобы не загромождать сам код программы множеством определений различных свойств компонентов, относящихся к визуальному представлению. Файлы, содержащие описание форм и их компонентов, создаются и изменяются в полностью автоматическом режиме, в зависимости от того, какие действия были проделаны в визуальной среде с данной формой.

Тем не менее, посмотреть и даже отредактировать исходный код такого файла можно – для этого достаточно щелкнуть по форме правой клавишей мышки и выбрать из контекстного меню пункт View as text. Это приведет к тому, что окно с формой закроется, а в редактор кода вместо файла с программным кодом будет загружен объектный код, являющийся описанием формы со всеми ее свойствами, объектами и

свойствами объектов. Вариант такого кода для формы с расположенной на ней кнопкой можно увидеть на листинге 11.1.

Листинг 11.1. Объектный файл формы

```
object Form1: TForm1
  Left = 192
  Top = 114
  Width = 482
  Height = 377
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 144
    Top = 64
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 1
  end
end
```

Исследовав подобные файлы, можно убедиться, что в них сохраняются все параметры, необходимые визуальному объекту для представления. В частности, в них хранятся размеры, параметры шрифтов, цвета, тексты надписей и т.д. В том случае, если вы назначите какие-либо дополнительные свойства, в том числе обработчики событий, то они так же будут отображены в таком файле.

Что касается возможности правки, то сразу следует отметить, что добавлять новые объекты путем помещения их описания в объектный файл – не лучшая идея. В то же время, использовать правку формы в текстовом виде может быть удобно, например, в том случае, если требуется изменить, скажем, какое-нибудь слово, которое повторяется во многих текстовых подписях на одной форме, или же передвинуть ряд перекрывающих друг друга компонентов, и т.д.

При всем этом следует учитывать, что правка исходного текста должна производиться с максимальной осторожностью, поскольку в том случае, если вы случайно повредите структуру этого файла, то это приведет к невозможности вернуться к визуальному режиму и продолжить работу над программой, содержащей испорченный таким образом модуль. Собственно для того, чтобы вернуться к обычному, визуальному, режиму работы над формой, следует вновь воспользоваться контекстным меню, только на этот раз не формы, а редактора кода, и выбрать из него пункт View As Form, или же нажать Alt+F12.

Что касается другой составляющей формы – файла pas, который мы, собственно и видим в паре с ее визуальным представлением, то в нем хранится описание формы как класса, а так же подпрограмм, типов и переменных, не являющихся частью формы как таковой, но по тем или иным соображениям помещенных в данный файл. Применительно к той же самой форме, содержащей лишь одну простую кнопку, то pas-файл для нее будет выглядеть так, как представлено на листинге 11.2.

Листинг 11.2. Программный файл формы

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;
type
  TForm1 = class(TForm)
    Button1: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}

end.
```

Мы видим, что модуль формы в части interface содержит включение целого ряда других модулей из библиотеки VCL, описание класса формы и глобальную переменную, представляющую собой форму как объект во время выполнения программы. В части implementation пока ничего нет, вернее, там имеется лишь ссылка на включение того самого объектного файла, который, с точки зрения Object Pascal, можно считать ни чем иным, как своеобразным конструктором формы.

Класс TForm

Итак, ключевую часть в программном модуле, описывающем то или иное окно приложения, является класс, описывающий это самое окно. Для форм в VCL, подобно другим компонентам, определен специальный класс – TForm. Однако следует учитывать, что на самом деле, класс TForm никогда напрямую не используется в приложениях: вместо него мы всегда будем работать с классами, порожденными на его основе. Дело в том, что формы всегда индивидуальны, ведь окна приложений содержат в себе другие элементы управления которые, в свою очередь, являются ни чем иным, как свойствами класса, описывающего форму. В частности, если вернуться к листингу 11.2, то видно, что форма в нем описывается как класс TForm1, а кнопка (Button1) является свойством этого класса. Если добавить к кнопке обработчик события (например, для OnClick), то этот обработчик автоматически получит имя Button1Click и будет определен как метод класса TForm1.

Разумеется, что при всем этом создаваемый класс, описывающий форму, унаследует все свойства и методы от базового класса TForm, который происходит от класса TCustomForm, добавляя к нему лишь методы, связанные с многооконным интерфейсом (MDI). Класс TCustomForm, в свою очередь, происходит от класса TScrollingWinControl, являющегося прямым потомком уже упоминавшегося ранее класса TWinControl, опять-таки, имеющего по отношению к своему родителю лишь одно глобальное отличие – возможность прокрутки содержимого. Учитывая все вышеизложенное, рассмотрим методы и свойства класса TCustomForm, которые достаточно часто используются применительно к создаваемым классам форм. В частности, среди методов следует отметить такие, как Close, Hide, Show, ShowModal, SetFocus и Release. Так, метод Close закрывает окно формы, но не удаляет ее из памяти. Правда, если это было единственным, или же главным окном приложения, то его закрытие приведет к завершению работы всего приложения.

ПРИМЕЧАНИЕ

> Каждое приложение в Windows имеет главное окно. Если окно единственное, то оно и будет главным, в том же случае, когда окон несколько, главным назначается то, которое будет создано первым. Порядок создания окон можно посмотреть на закладке Forms окна свойств проекта (Project → Options), или же непосредственно в файле проекта. При необходимости этот порядок всегда можно изменить.

Методы Show и Hide делают окно видимым и невидимым, соответственно. А метод ShowModal делает окно не только видимым, но и единственным доступным для приложения, закрывая за собой все остальные окна – модальным. Типичный пример модального окна – диалог подтверждения при выходе из программы (рис11.5).

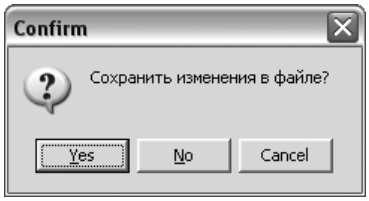

Рис. 11.5. Типичный модальный диалог

Метод SetFocus устанавливает фокус ввода на тот или иной элемент управления, находящийся на форме. Ну и, наконец, метод Release является специализированным деструктором форм, т.е., если для обычных классов используется метод Destroy, а для компонент – Free, то для окон – Release.

Что касается свойств класса TForm, то все они являются унаследованными от родительских классов. Те их них, что принадлежат к еще не рассмотренному нами классу TCustomForm, представлены в таблице 11.1.

Таблица 11.1. Основные свойства класса TCustomForm

Свойство	Описание
Active	Указывает, имеет ли в данный момент форма фокус ввода
ActiveControl	Указывает на компонент, имеющий фокус ввода на форме
BorderIcons	Определяет, какие кнопки должны быть на системном заголовке окна. Допустимые значения: biSystemMenu, biMinimize, biMaximize, biHelp

BorderStyle	Определяет вид и возможности рамки окна. Допустимые значения: bsNone, bsSingle, bsSizeable, bsDialog, bsToolWindow, bsSizeToolWin
Canvas	Предоставляет доступ к холсту окна
ClientHeight	Указывает высоту клиентской части окна в пикселях (клиентская область исключает рамки, заголовок, полосы прокрутки и т.д.)
ClientRect	Указывает габариты прямоугольника, образующего клиентскую часть окна
ClientWidth	Указывает ширину клиентской части окна в пикселях
FormState	Указывает на текущее состояние формы. Допустимые значения: fsCreating, fsVisible, fsShowing, fsModal, fsCreatedMDIChild, fsActivated
FormStyle	Определяет тип формы. Допустимые значения: fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop
HelpFile	Определяет файл справки, предназначенный для данной формы
Icon	Определяет иконку формы
Menu	Определяет компоненту-главное меню окна
ModalResult	Используется для возвращения результата из модальных диалоговых окон
Position	Определяет размеры и расположение окна. Допустимые значения: poDesigned, poDefault, poDefaultPosOnly, poDefaultSizeOnly, poScreenCenter, poDesktopCenter, poMainFormCenter, poOwnerFormCenter
Visible	Определяет, должно ли окно быть видимым
WindowState	Определяет, в каком виде должно появиться окно на экране. Допустимые значения: wsNormal, wsMinimized, wsMaximized

Рассмотрим подробнее некоторые из приведенных в таблице 11.1 свойств. В частности, свойство BorderIcons отвечает за то, какие системные кнопки будут доступны на заголовке окна. В типичном случае, для стандартного вида окна, используются 3 кнопки – сворачивания в панель задач (biMinimize), разворачивания на весь экран (biMaximize) и кнопка системного меню, отображающаяся в паре с кнопкой закрытия приложения. И хотя в Delphi предоставляется выбор каждой кнопки в отдельности, следует помнить об ограничениях, накладываемых самой Windows. Прежде всего, если попытаться удалить кнопку системного меню (выбрав для biSystemMenu значение false), то из заголовка исчезнут вообще все кнопки. Другое ограничение связано с кнопками сворачивания и разворачивания: если попытаться убрать только одну из них, то вторая все равно останется, однако будет неактивна.

ПРИМЕЧАНИЕ

> Все свойства, связанные с отображением и типом формы, в частности, для рассматриваемых кнопок системного меню, а так же BorderStyle, FormStyle, Position, Visible и WindowState, проявляют себя только во время выполнения программы.

Еще одно важное замечание по свойству BorderIcons состоит в том, что на него имеет непосредственное влияние значение свойства BorderStyle. Например, кнопка контекстной справки (biHelp) будет видна только в том случае, если в качестве значения BorderStyle установлено bsDialog. Однако при этом кнопки сворачивания и разворачивания не будут отображены, вне зависимости от того, что установлено для них в свойстве BorderIcons. Ну а если свойство BorderStyle имеет значение bsNone, то вообще не только никаких кнопок, но и самого заголовка окна выведено не будет. В

целом эффект установки значений для свойства BorderStyle и его воздействие на некоторые другие свойства формы, рассмотрены в таблице 11.2.

Таблица 11.2. Свойство BorderStyle

Значение	Описание	Воздействие на другие свойства
bsNone	Системный заголовок окна и рамка отсутствуют	Установка BorderIcons не имеет смысла
bsSingle	Одинарная рамка окна, не позволяющая изменять его размеры	Для BorderIcons нельзя установить biHelp
bsSizeable	Стандартное изменяемое окно	Для BorderIcons нельзя установить biHelp
bsDialog	Окно типа диалога. Рамка есть, но изменить размеры окна, свернуть его или развернуть нельзя	Для BorderIcons нельзя установить biMinimize и biMaximize. Свойство MainMenu не имеет смысла
bsToolWindow	Аналогично bsSingle, но с маленьким заголовком	Отображается только кнопка закрытия окна
bsSizeToolWin	Аналогично bsSizeable, но с маленьким заголовком	Отображается только кнопка закрытия окна

Свойство Position отвечает за то, в каком месте экрана появится окно. Его возможные значения и их описания приведены в таблице 11.3.

Таблица 11.3. Свойство Position

Значение	Описание
poDesigned	Окно появится на экране в том же месте и при тех же размерах, что во время разработки в среде Delphi
poDefault	Размеры и расположение окна определяются операционной системой. При каждом запуске окно будет сдвигаться немного вниз и вправо
poDefaultPosOnly	Расположение окна будет определяться операционной системой, а его размеры будут такими же, как во время разработки
poDefaultSizeOnly	Размеры окна будет определяться операционной системой, а его расположение будет таким же, как во время разработки
poScreenCenter	Размеры окна будут такими же, как при разработке, а его расположение будет выровнено по центру экрана
poDesktopCenter	Аналогично psScreenCenter, но для приложений, рассчитанных на несколько экранов, выбор главного монитора не будет произведен
poMainFormCenter	Размеры окна будут такими же, как при разработке, а его расположение будет выровнено по центру главного окна приложения. Данное значение используется только для вторичных форм приложения
poOwnerFormCenter	Размеры окна будут такими же, как при разработке, а его расположение будет выровнено по центру окна-владельца, задаваемого при помощи свойства Owner

Еще одно свойство, отвечающее за появление окна на экране – WindowState - определяет, в каком состоянии оно должно появиться. При этом данное свойство действует «с оглядкой» на BorderStyle: если при выбранном для BorderStyle значении окно не может быть свернуто или развернуто, то установка соответствующих значений для WindowState так же не возымеет силы.

Наконец, такие свойства, как Active, ActiveControl и FormState, являются доступными только во время выполнения программы и могут быть использованы для определения текущего состояния окна.

Приложения SDI и MDI

При рассмотрении класса TForm мы намеренно опустили ряд свойств и методов, относящихся к многодокументному интерфейсу приложения. Дело в том, что прежде следует разобраться с самими принципами построение интерфейса приложений в ОС Windows.

Исторически сложилось так, что все Windows-приложения классифицируются по двум типам организации оконного интерфейса – SDI (Single Document Interface – однодокументный интерфейс) и MDI (Multiple Document Interface – многодокументный интерфейс). Во времена Windows 3.x Microsoft продвигала многодокументные версии приложений (начиная от диспетчера программ Windows и заканчивая офисными программами, например, Word for Windows). Но с момента появления Windows 95 сначала Microsoft, а затем и другие разработчики стали все чаще склоняться к однодокументной компоновке программ. Судя по всему, это было вызвано тем, что неподготовленному пользователю сложнее разобраться с «окнами в окнах» в стиле MDI. Вместе с тем, оба подхода к организации приложения имеют право на существование, поэтому рассмотрим вопросы организации MDI-приложений.

Итак, программы с однодокументным интерфейсом имеют одно или несколько окон, которые можно перемещать, открывать и закрывать вне зависимости друг от друга. Большинство современных приложений (скажем, Delphi) реализуют SDI-интерфейс: вы можете открыть несколько документов, при этом каждый из них будет находиться в отдельном, вполне самостоятельном окне.

В противоположность такому подходу, приложения с многодокументным интерфейсом имеют одно главное окно, а все остальные документы открываются внутри этого окна. Из примеров такого подхода среди современных программ можно привести, пожалуй, браузер Opera, хотя реализация MDI в последних версиях этого браузера несколько закамуфлирована.

В Delphi поддерживаются оба типа приложений, причем если по умолчанию создаются SDI-программы, то для создания MDI-приложения можно либо использовать заготовку, либо сделать главную (первую) форму приложения MDI-контейнером. Первый способ кажется быстрее: достаточно в меню File → New выбрать пункт Other, а в группе Projects открывшегося окна New Items щелкнуть по MDI Application и выбрать расположение проекта на диске. Результатом будет загрузка в рабочую среду Delphi нового приложения на основе шаблона многодокументного интерфейса, подготовленного специалистами Borland.

В итоге мы получим почти полноценное MDI-приложение, которое умеет создавать новые окна для своих «документов», открывать в них текстовые файлы и закрывать их, а так же упорядоченно размещать все дочерние окна в своей рабочей области (рис. 11.6). Фактически, приложив совсем немного усилий, на основе этой заготовки можно (но, откровенно говоря, вряд ли действительно не нужно!) создать очередной «улучшенный» вариант Блокнота (notepad.exe) на основе многодокументного интерфейса.

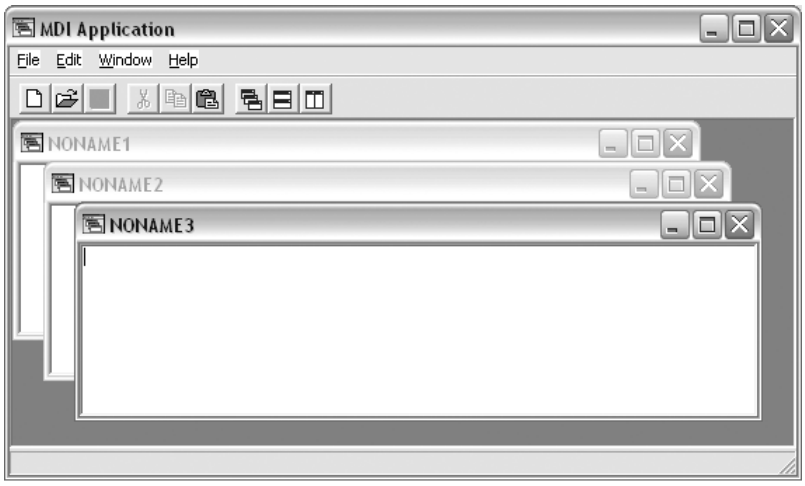

Рис. 11.6. Приложение с MDI на основе шаблона Delphi

Другой способ создать MDI-приложение заключается в том, что потребуется установить свойство FormStyle главного окна приложения в fsMDIForm. После этого можно создать еще одну форму (File → New → Form) и для нее установить это же свойство в значение fsMDIChild. Таким образом, первая форма станет главным окном, или MDI-контейнером, а вторая – дочерним окном.

Помимо свойства FormStyle, за реализацию MDI отвечают еще несколько свойств и методов класса TForm. Среди свойств это ActiveMDIChild, указывающее, какое из дочерних окон имеет фокус ввода, MDIChildCount, указывающее на количество открытых дочерних форм, а так же MDIChildren, предоставляющий доступ ко всем дочерним окнам. Последнее свойство представляет собой массив, ссылающийся на все дочерние окна, первое из них имеет индекс 0, а последнее – на единицу меньшее, чем их общее количество (т.е. MDIChildCount-1).

Кроме того, есть еще одно свойство – TileMode, определяющее, каким образом окна должны быть расположены при вызове метода Tile. Это свойство может принимать значения tbHorizontal – для горизонтального размещения, и tbVertical – для вертикального. Сам метод Tile, соответственно, используется для размещения дочерних окон в родительском на основе значения, заданного для свойства TileMode.

Другие методы формы, относящиеся к MDI-приложениям, это ArrangeIcons, Cascade, Next и Previous. Метод ArrangeIcons позволяет упорядочить все свернутые окна так, чтобы они расположились в линии и не перекрывали друг друга. Метод Cascade, наоборот, действует на раскрытые окна: обращение к этому методу расставляет окна каскадом, одно за другим со смещением вниз и влево (см. рис. 11.6). Что касается методов Next и Previous, то они, соответственно, позволяют переключаться к следующему и предыдущему дочернему окну.

При разработке MDI-приложений следует учитывать, что рабочая область главного окна не должна «загромождаться» какими-либо элементами управления. Единственное, что можно на нем разместить – это главное меню и инструментальную панель. Что касается дочерних окон, то на них, наоборот, не желательно иметь ни того, ни другого.

Стандартные компоненты Delphi

От форм, являющихся основой приложений, перейдем к деталям, т.е. к тому, что на этих формах располагается. Начнем с компонент, расположенных на закладке Standard – наиболее часто используемых в Windows-приложениях. Это текстовые надписи, кнопки, переключатели, текстовые поля, панели и иные повсеместно используемые элементы пользовательского интерфейса.

Кнопки

Пожалуй, самым важным элементом управления в любой программе является кнопка, ведь, фактически, без нажатия на кнопку не начинается ни одно действие в Windows-приложениях. Для создания этого элемента управления в VCL предусмотрено несколько компонент, но наиболее востребованным является Button (кнопка). Он находится на закладке Standard палитры компонентов и выглядит как маленькая кнопка с надписью «OK».

Компонент Button достаточно простой, и в дополнение к унаследованным от TWinControl свойствам и методам, имеет лишь 5 собственных свойств. Помимо свойства Caption, определяющего отображаемый на кнопке текст, и WordWrap, отвечающего за возможность переноса слов в случае, если надпись не помещается в одну строку, у компонента Button имеются такие специфические свойства, как Cancel и Default. Их смысл заключается в следующем:

- Cancel – если это свойство установлено в истину, то нажатие на клавишу Esc будет равнозначно щелчку по этой кнопке. Как правило, это значение устанавливают в истину для кнопки «Отмена»;
- Default – установка в истину данного свойства делает кнопку восприимчивой к нажатию Enter, даже если она не имеет фокуса ввода. Это бывает полезным для кнопок подтверждения, например, «OK».

Здесь следует заметить, что, во-первых, на форме должна быть только одна кнопка типа отмены и одна – типа подтверждения. Кроме того, хотя явных ограничений VCL и не накладывает, по своей логике эти свойства являются взаимоисключающими.

Наконец, последнее свойство – ModalResult, имеет смысл использовать в том случае, если кнопка помещена на модальную диалоговую форму. В таком случае, если кнопке присвоено значение ModalResult, отличное от mrNone, то нажатие на такую кнопку приведет к закрытию диалогового окна и возвращению вызвавшей его процедуре значения данной кнопки.

Для примера создадим проект, состоящий из главной формы, на которую следует поместить кнопку. После этого добавим к проекту еще одну форму (File → New → Form), на которую, в свою очередь, поместим уже 2 кнопки. Левую сделаем кнопкой по умолчанию, возвращающую результат подтверждения. Для этого надо выбрать эту кнопку на форме и произвести в инспекторе объектов следующие действия:

- В свойстве Caption написать строку «OK»;
- Свойство Default установить в true;
- Для свойства ModalResult выбрать значение mrOk.

Для второй кнопки следует сделать подобные операции, со следующими отличиями:

- В Caption написать «Отмена»;
- Свойство Cancel установить в true (свойство Default оставить false);
- Для свойства ModalResult выбрать значение mrCancel

Теперь можно щелкнуть по самой форме, и в инспекторе объектов выбрать для ее свойства Position значение poMainFormCenter, поскольку для модельных диалоговых окон характерно появление как раз по центру главного окна приложения. Так же не помешает установить свойство BorderStyle в bsDialog. Наконец, размеры формы желательно сделать не очень большими, поскольку все равно кроме этих 2 кнопок не ней ничего не будет (рис. 12.1), а в качестве заголовка можно написать «Модальное окно». На этом работу над этой формой можно считать завершенной.

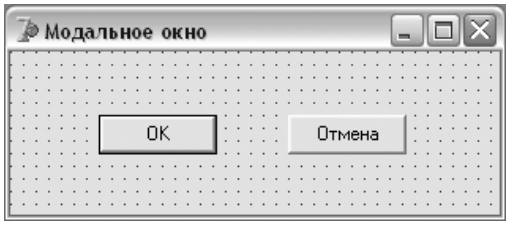

Рис. 12.1. Форма модального диалогового окна

Сохраним проект, согласившись с предложенными по умолчанию именами (Unit2, Unit1 и Project1) в какой-либо папке на жестком диске и вернемся к первой, главной форме программы. На ней уже имеется кнопка, так что теперь следует написать код для обработчика события щелчка мышью. Для этого щелкните по кнопке мышкой и выберите на закладке Events инспектора объектов событие onClick и дважды щелкните по полю напротив, чтобы Delphi подготовила необходимый программный код.

СОВЕТ

> На самом деле, для того, чтобы создать обработчик события onClick для кнопки достаточно дважды щелкнуть по самой кнопке. Двойной щелчок по визуальному компоненту, помещенному на форму, создает обработчик для наиболее характерного события, связанного с данным элементом управления. Раузметтся, для кнопки это как раз и будет onClick.

В результате вы получите заготовку для обработчика события onClick этой кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

В него нам надо поместить код, который будет показывать вторую форму как модальное диалоговое окно. Но прежде, чем обратиться к ней, следует позаботиться о том, чтобы модуль второй формы был доступен в модуле первой. этого чуть выше созданного обработчика событий, на непосредственно после строки с ключевым словом «implementation», следует написать:

```
uses unit2;
```

Здесь следует отметить тот момент, что хотя в данном модуле уже имеется часть uses, помещенная в самом начале, вслед за ключевым словом interface, включать другие формы программы туда не рекомендуется во избежание возникновения перекрестных связей. Они могут возникнуть, например, если не только первая форма обращается ко второй, но и вторая – к первой.

Теперь все готово к написанию обработчика события onClick. Прежде всего, он должен показывать диалоговую форму в модальном режиме. Для этого достаточно написать такую строку кода:

```
Form2.ShowModal;
```

Вслед за ним разместим код, который будет выводить в заголовок этого окна результат, полученный от вызванного диалога:

```
if Form2.ModalResult = mrOk then Form1.Caption:='OK';
if Form2.ModalResult = mrCancel then Form1.Caption:='Cancel';
```

Отметим, что после обращения к методу ShowModal управление будет передано вызванной таким образом форме. Соответственно, пока пользователь не закроет тем или иным способом модальное окно, следующий оператор выполнен не будет. Пример можно посмотреть в папке Demo\Part3\Button.

Надписи

Надписи, или текстовые метки (labels), часто используются в программах для вывода пояснительной информации. В VCL для этих целей используется компонент TLabel. На палитре компонентов он расположен 4-м слева, в виде кнопки с буквой «A».

ПРИМЕЧАНИЕ

> Самой первой на любой закладке палитры компонент является кнопка со стрелкой, которая является не компонентой, а инструментом IDE. Ее следует использовать в том случае, если вы выберите компонент, но передумаете размещать его на форме. В таком случае щелчок по этой кнопке сбросит выбранный компонент.

Текст, помещенный на форму при помощи меток, пользователю нельзя редактировать. Кроме того, такой элемент не может даже получить фокус ввода. Это объясняется тем, что класс TLabel происходит не от оконных элементов интерфейса (TWinControl), а от более легких графических (TGraphicControl). От этого предка классу TLabel достается свойство Canvas. Что касается остальных свойств метки, то помимо всех унаследованных, включая самое важное для метки свойство – Caption, отвечающее за собственно надпись, она имеет несколько специфических свойств, отвечающих за выравнивание и расположение. Все эти свойства приведены в таблице 12.1.

Таблица 12.1. Свойства TLabel

Свойство	Тип значения	Описание
Alignment	TAlignment	Определяет выравнивание текста по горизонтали в области метки. Может принимать значения taLeftJustify, taRightJustify и taCenter
AutoSize	Boolean	Определяет, должна ли метка изменять свои размеры в зависимости от текста
FocusControl	TWinControl	Указывает на элемент управления, который может быть ассоциирован с меткой

Layout	TTextLayout	Определяет выравнивание текста по вертикали. Может принимать значения tlTop, tlCenter и tlBottom
ShowAccelChar	Boolean	Определяет, должен ли символ & обозначать подчеркнутую букву. Если да, то для вывода самого символа & его надо будет указать дважды (&&)
Transparent	Boolean	Определяет, должен ли фон метки быть прозрачным
WordWrap	Boolean	Определяет, должен ли текст, не помещающийся по ширине, переноситься на следующие строки

При установке свойств выравнивания следует учитывать, что такие свойства, как Alignment и Layout имеют смысл лишь в том случае, если свойство AutoSize не установлено в истину, поскольку в противном случае границы надписи всегда будут определяться лишь текстовым содержимым. А свойство ShowAccelChar следует использовать лишь в паре с FocusControl, поскольку, по правилам интерфейса Windows, подчеркнутая буква означает, что при нажатии ее на клавиатуре совместно с клавишей Alt, ассоциированный с надписью элемент управления получит фокус ввода.

Вместе с тем, в Delphi имеется и другой аналогичный по своему назначению компонент – StaticText, расположенный на закладке Additional палитры компонентов. Его отличие от Label состоит в том, что он является потомком класса TWinControl, со всеми вытекающими отсюда последствиями, а именно – наличия собственного дескриптора оконного элемента (Handle), возможности быть заключенным в рамку (свойство BorderStyle) и т.д. Но в типичном случае все-таки предпочтительнее использовать менее ресурсоемкий элемент Label.

Поля ввода текста

Вслед за меткой на палитре компонент располагается другой часто используемый в интерфейсе Windows-приложений элемент управления, Edit – строка ввода текста. Этот компонент, в отличие от метки, является полноценным оконным элементом управления, происходящим от TWinControl, и может не только получать фокус ввода, но и использоваться для правки однострочного текста пользователем. При этом поддерживаются такие операции, как перемещение по строке при помощи клавиш управления курсором, удаление символов, выделение текста, в том числе при помощи мышки, а так же взаимодействие с буфером обмена Windows (clipboard).

Компонента Edit имеет ряд свойств и методов, специфических для элементов редактирования текста. Все они инкапсулированы в классе TCustomEdit, потомками которого, помимо TEdit, являются и другие элементы интерфейса, связанные с вводом текста. Что касается его свойств, то они приведены в таблице 12.2.

Таблица 12.2. Свойства TCustomEdit

Свойство	Тип значения	Описание
AutoSelect	Boolean	Определяет, должен ли быть выделен весь текст при получении фокуса ввода
AutoSize	Boolean	Определяет, должна ли автоматически изменяться высота элемента при изменении размера шрифта
BorderStyle	TBorderStyle	Определяет, должен быть компонент обрамлен рамкой (bsSingle), или нет (bsNone)
CanUndo	Boolean	Указывает, может ли в данный момент быть применен

		метод Undo
CharCase	TEditCharCase	Определяет, должен ли вводимый текст преобразовываться в символы верхнего или нижнего регистра. Допустимые значения: ecNormal, ecUpperCase и ecLowerCase
HideSelection	Boolean	Определяет, должно ли сохраняться визуальное выделение текста при потере фокуса ввода
MaxLength	Integer	Определяет максимально допустимое количество символов в строке
Modified	Boolean	Указывает, был ли текст изменен пользователем
PasswordChar	Char	Определяет символ, который должен отображаться вместо вводимых символов
ReadOnly	Boolean	Определяет, может или нет пользователь редактировать текст
SelLength	Integer	Указывает на длину выделенного фрагмента текста
SelStart	Integer	Указывает на позицию первого выделенного символа
SelText	String	Содержит строку с выделенным фрагментом текста

Из свойств, влияющих на внешнее оформление компоненты, отметим AutoSize и BorderStyle. Первое, в отличие от одноименных свойств метки, в данном случае изменяет размеры компоненты только по высоте. А второе, на сей раз – в отличие от такого же свойства у формы, может принимать только 2 значения, которые сводятся к тому, будет рамка вообще или нет.

Все остальные свойства, так или иначе, связаны непосредственно с текстовым содержимым. Например, если установить PasswordChar в значение, отличное от нулевого символа (#0), что принято по умолчанию, то мы получим элемент управления, предназначенный для ввода паролей. Как правило, в этом случае для замены используют символ «звездочка». Разумеется, при этом вводимые символы не будут в действительности заменяться, а останутся теми, что были введены в действительности.

Зато другое свойство, влияющее на ввод текста – CharCase – действительно может влиять на вводимые символы. При этом если оно будет установлено в значение ecUpperCase, то все вводимые буквы станут преобразовываться в символы верхнего регистра, а если в ecLowerCase – то в нижнего.

Свойство MaxLength, при значении, большим нуля, ограничивает длину текста. А свойство ReadOnly, будучи установленным в истину, вообще запретит правку текста пользователем. Если же правка возможна, то при помощи свойства Modified можно узнать, изменял ли пользователь текст: оно автоматически устанавливается в истину при любом изменении содержимого, кроме тех, что были произведены программными методами. Ну а если какая-либо правка была произведена, то возможен и возврат к предыдущему состоянию – за этим «следит» свойство CanUndo.

Все оставшиеся свойства связаны с выделением текста. Как известно, в Windows текст выделяется либо клавиатурой (перемещением с нажатой клавишей Shift), либо мышкой. Разумеется, компоненты VCL, являясь, по большей части, оболочкой для системных средств ОС, поддерживают оба этих метода и предоставляют разработчику все необходимое для работы с выделенным фрагментом. В частности, первый выделенный символ можно определить по свойству SelStart. Если же ничего не выделе-

но, то это свойство будет ссылаться на текущую позицию каретки в строке. Но если текст все-таки выделен, то длину выделенного фрагмента можно узнать по текущему значению свойства SelLength. Просуммировав оба этих значения, при желании, можно определить и последний выделенный символ:

```
SelEnd := Edit1.SelStart + Edit1.SelLength;
```

Оба этих свойства можно менять программно, выделяя нужную часть текста. Сам же выделенный фрагмент текста можно получить, обратившись к свойству SelText.

Установив свойство AutoSelect в ложь, можно сделать так, что при получении фокуса ввода текст в поле ввода не станет выделенным автоматически (это не актуально, если пользователь устанавливает фокус ввода, пользуясь мышкой). А установив свойства HideSelection в ложь, можно получить такой эффект,, что даже после потери фокуса ввода выделенный фрагмент текста не потеряет выделения.

Наглядную иллюстрацию этих свойств можно получить, экспериментируя с настройками этой компоненты. Ряд вариантов приведен в примере editprops, находящемся в каталоге Demo\Part3\Edit.

Разобравшись со свойствами, перейдем к методам. Все они также являются унаследованными от класса TCustomEdit, и предназначены для работы с выделенным текстом, включая взаимодействие с буфером обмена. Так же есть методы для работы с содержимым в целом. Так, для удаления всего текста используют метод Clear, для выделения всего содержимого – метод SelectAll, а для сброса выделения – метод ClearSelection.

Для такой незаменимой операции, как отмена, применяются методы Undo и ClearUndo. При этом собственно отмену (при ее возможности, для чего следует проверять свойство CanUndo) производят при помощи метода Undo. Второй же метод позволяет сбросить список произведенных действий, делая отмену невозможной. Например, реализация разовой отмены может выглядеть следующим образом:

```
if Edit1.CanUndo then begin
  Edit1.Undo;
  Edit1.ClearUndo;
end;
```

Оставшиеся методы – CopyToClipboard, CutToClipboard и PasteFromClipboard используются для таких стандартных операций с буфером обмена, как копирование, вырезание и вставка. При этом копирование и вырезание возможно лишь в том случае, если имеется выделенный текст. Таким образом, перед обращением к этим методам следует проверять, есть ли выделенный фрагмент текста. Делать это можно, проверяя значение свойства SelLength:

```
if Edit1.SelSength > 0 then ...
```

Что касается операции вставки, то она, разумеется, возможна только в том случае, если в буфере обмена находится текст. Если находящийся в буфере текст состоит более чем из одной строки (т.е. разделен символами «новая строка»), то в поле редактирования будет вставлена лишь первая строка текста.

Многострочный редактор и строки

Для работы с многострочным текстом используется другой наследник класса TCustomEdit – компонент Memo, или многострочный редактор. На палитре компонентов его значок расположен сразу после однострочного редактора. Класс TMemo имеет все свойства однострочного редактора, а так же ряд собственных свойств, необходимых учитывающих специфику многострочного редактирования. Прежде всего, это свойство WordWrap, отвечающее за автоматический перенос не умещающихся по ширине окна строк текста. А свойство Alignment отвечает за выравнивание текста по левому (taLeftJustify) или правому (taRightJustify) краям, или по центру (taCenter).

Еде для работы с множеством строк могут понадобиться полосы прокрутки. За их наличие и расположение отвечает свойство ScrollBars, которое может принимать одно за 4 значений: ssNone – без полос прокрутки (это значение принято по умолчанию), ssHorizontal – полоса для горизонтальной прокрутки, ssVertical – полоса для вертикальной прокрутки и ssBoth – с обеими полосами прокрутки. При этом отсутствие полос не ограничивает размер вводимого текста, хотя и влияет на его ввод.

Так, если установлена горизонтальная прокрутка, то вводимый текст не будет переноситься, вне зависимости от того, какое значение имеет свойство WordWrap. В то же время, если полосы прокрутки отсутствуют вообще, то при включенном переносе слов, вводимый текст будет ограничиваться по видимой ширине редактора, а при выключенном – уходить вправо, сдвигая видимую границу за левый край окна.

Еще 2 свойства отвечают за обработку отдельных клавиш на клавиатуре. Это WantTabs и WantReturns. Первое отвечает за обработку нажатий клавиши табуляции, а второе – клавиши ввода. При этом значение истины для обоих свойств означает, что эти клавиши будут обрабатываться самим многострочным редактором. Т.е. нажатие на Tab будет вставлять символ табуляции, а нажатие на Enter – к началу новой строки. Если же значение этих свойств будет ложью, то нажатие на Tab приведет к перемещению фокуса ввода к следующему элементу управления, а обработка нажатия клавиши Enter будет передана форме. По умолчанию свойство WantTabs установлено в false, а WantReturns – в true.

Для того чтобы определить, где в данный момент находится каретка, обращаются к свойству CaretPos. Оно возвращает координаты (X, Y) точки по отношению к верхнему левому углу окна редактора. Если же необходимо определить смещение не в пикселях от угла компоненты, а в символах от начала строки, то следует использовать свойство SelStart.

Наконец, рассмотрим свойства, относящиеся собственно к текстовому содержимому многострочного редактора. Разумеется, у него есть свойство Text, через которое можно получить доступ к всему содержимому сразу. Но если вы поместите элемент Memo на форму и посмотрите на инспектор объекта, то не найдете в нем этого свойства. Дело в том, что оно не является опубликованным для класса TMemo, и доступно лишь программно. Зато у многострочного редактора имеется другое, более подходящее свойство – Lines, являющееся опубликованным.

Если в инспекторе объекта щелкнуть по кнопке с многоточием напротив этого свойства, то откроется окно редактирования, в котором можно ввести начальный текст (рис. 12.2).

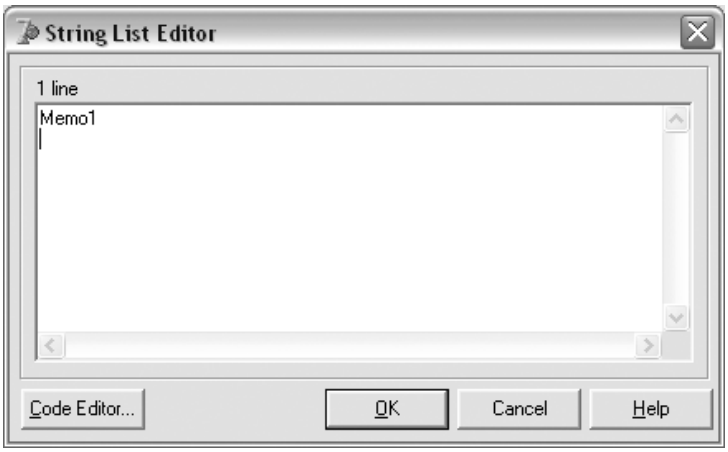

Рис. 12.2. Текстовый редактор для многострочных компонент в Delphi IDE

Поскольку это окно само фактически, именно этот компонент там и использует, то в нем можно ввести в точности то, что должно, по замыслу разработчика, находиться в текстовом окне программы.

В качестве типа данных для свойства Lines используется не строка, а специальный объект TStrings, являющийся классом, инкапсулирующем работу со строками, или, если быть точнее – со списками строк. С его помощью можно манипулировать содержимым многострочного редактора, рассматривая его как отдельные строки. Свойства класса TStrings приведены в таблице 12.3.

Таблица 12.3. Свойства TStrings

Свойство	Тип данных	Описание
CommaText	String	Представляет весь список одной строкой, разделенной запятыми. При этом строки с пробелами заключаются в двойные кавычки
Count	Integer	Указывает на количество строк в списке
DelimitedText	String	Представляет весь список одной строкой в соответствии со значениями свойств Delimeter и QuoteChar
Delimiter	Char	Определяет разделитель для свойства DelimitedText
Names	[Index: Integer]: string;	Представляет собой массив имен для строк формата имя-значение
NameValueSeparator	Char	Определяет символ, по которому будут разделены имя и значение
Objects	[Index: Integer]: TObject;	Представляет собой массив объектов, ассоциированных со строками
QuoteChar	Char	Определяет символ кавычек для свойства DelimitedText

Strings	[Index: Integer]: string;	Представляет список в качестве массива строк
Text	String	Список строк одной строкой, включая символы «возврат каретки» и «новая строка»
ValueFromIndex	[Index: Integer]: string;	Представляет собой массив значений для строк формата имя-значение
Values	[const Name: string]: string;	Позволяет получить значение по его имени для строк формата имя-значение

Все свойства этого класса можно разделить на 2 категории: те, что предназначены для работы со строками списка по отдельности, и те, что позволяют взаимодействовать со всем текстом в целом. К первой группе относятся свойства Count, Strings, Names и Values. Причем последние 2 свойства используются при работе со списком, состоящим из строк типа имя значение, например, «город=Москва». К этой группе можно причислить свойства NameValueSeparator и ValueFromIndex. Для примера рассмотрим список, состоящий из строк «город=Москва» и «страна=Россия». Чтобы можно было рассматривать этот список как состоящий из пар имя-значение, следует назначить свойству NameValueSeparator значение «=». После этого можно обращаться к свойствам Names, Values и ValueFromIndex. Например, применительно к строкам, содержащимся в многострочном редакторе (пусть он называется Memo1), мы можем использовать подобный код (см. также пример в Demo\Part3\Memo):

```
var s: string;
...
s := Memo1.Lines.Names[0]; // s получит «Москва»
Memo1.Lines.ValueFromIndex[0]:='Подольск';
s := Memo1.Lines.Values['город']; // s получит «Подольск»
s := Memo1.Lines.Values['страна']; // s получит «Россия»
```

Если же предстоит работать с обычными строками, то можно использовать свойство Strings. Так, для обращения к первой строке списка достаточно написать:

```
s := Memo1.Lines.Strings[0];
Memo1.Lines.Strings[1]:='Вторая строка';
```

Несколько особняком стоит свойство Objects. Дело в том, что список строк позволяет ассоциировать какой-либо объект с каждой строкой. В ряде случаев это оказывается очень полезным дополнением.

Что касается свойств, предназначенных для работы со списком как с единой строкой текста, то это, прежде всего, свойство Text, представляющее все содержимое списка в качестве одной строки. При этом в тех местах, где заканчиваются реальные строки списка, добавляются символы новой строки и возврата каретки (#10 и #13).

ПРИМЕЧАНИЕ

> Применительно к компоненту Memo, его собственное свойство Text содержит точно те же данные, что и Lines.Text.

Другое свойство, CommaText, также представляет весь список одной строкой, но использует для этого формат SDF, когда каждая строка перечисляется через запятую.

При этом если исходная строка не содержит ни запятых, ни пробельных символов, то она остается как есть, в противном случае – заключается в кавычки. Например, список, состоящий из строк «Borland Delphi» и «Forever», преобразуется в строку «"Borland Delphi",Forever».

Еще одно свойство, DelimetedText, является во многом аналогичным CommaText, за тем исключением, что можно выбирать тип кавычек и разделителей, которые, в свою очередь, определяются свойствами QuoteChar и Delimiter. Таким образом, если эти свойства установить в «"» и «,», то значение DelimetedText будет полностью соответствовать значению CommaText.

Что касается методов класса TStrings, то среди них можно выделить Add, AddObject, AddStrings, Append, Assign, Clear, Delete, IndexOf, Insert, LoadFromFile и SaveToFile. Пожалуй, самыми интересными являются последние 2 метода, позволяющие без каких-либо дополнительных действий загрузить из файла или сохранить в файл сразу все содержимое текстового редактора. Более подробно все эти методы будут рассмотрены в рамках класса TStringList.

Класс TStringList

Хотя в качестве свойств ряда компонент в VCL выступают объекты типа TStrings, использовать объекты этого класса сами по себе нельзя, поскольку этот класс является абстрактным. Соответственно, для работы непосредственно со списками строк в программах надо либо создавать классы, основанные на TStrings, либо использовать его наследника, «готового к употреблению» – класс TStringList.

Помимо того, что класс TStringList не является виртуальным, он добавляет функциональность к своему предшественнику, а именно – возможность поиска и сортировки строк в списке. В частности, он имеет такие методы, как CustomSort, Find и Sort, а также связанные с сортировкой свойства:

- CaseSensitive – это свойство определяет, долен ли учитываться символ регистра при сравнении во время поиска совпадений и сортировки;
- Duplicates – это свойство определяет, разрешено или нет помещать в список одинаковые строки;
- Sorted – данное свойство определяет, надо ли производить сортировку сразу при добавлении новых строк.

Если свойства CaseSensitive и Sorted могут принимать всего по 2 значения – ложь или истина, то для свойства Duplicates предусмотрено 3 варианта, определяющих поведение списка при добавлении новых строк, совпадающих с уже присутствующими в списке. Так, если для него установлено принятое по умолчанию значение dupAccept, то одинаковые строки будут добавляться, а если установлено dupIgnore – то не будут. Если же установить его в dupError, то попытка добавить к списку повторяющуюся строку будет вызывать исключение типа EStringListError.

Собственно для добавления новых строк к списку используют методы Add, Append или AddObject, а если требуется добавить сразу группу строк, т.е. другой список – то AddStrings. Эти, а так же другие методы классов TStrings и TStringList представлены в таблице 12.4.

Таблица 12.4. Методы списков строк

Метод	Параметры	Описание
Add	const S: string	Добавляет новую строку к списку
AddObject	const S: string; AObject: TObject	Добавляет строку к списку и ассоциирует с ней объект
AddStrings	Strings: TStrings	Добавляет группу строк к списку
Append	const S: string	Добавляет строку к списку, но в отличие от Add не возвращает результат операции
Assign	Source: TPersistent	Присваивает строки, и, по возможности, объекты, из другого списка к данному
Clear	-	Удаляет все строки списка
CustomSort	Compare: TStringList-SortCompare	Сортирует строки в списке используя правила, определенные в указанной функции
Delete	Index: Integer	Удаляет из списка строку с указанным индексом
Equals	Strings: TStrings	Сравнивает 2 списка и возвращает результат сравнения
Exchange	Index1, Index2: Integer	Меняет между собой расположение 2 строк в списке
Find	const S: string; var Index: Integer	Находит строку в отсортированном списке и возвращает ее индекс
IndexOf	const S: string	Аналогично Find, но подходит и для несортированных списков
IndexOfName	const Name: string	Возвращает индекс первого найденного совпадения имени для списков типа имя-значение
IndexOfObject	AObject: TObject	Возвращает индекс первой строки, ассоциированной с указанным объектом
Insert	Index: Integer; const S: string	Вставляет строку в указанную позицию списка
InsertObject	Index: Integer; const S: string; AObject: TObject	Вставляет строку и объект в указанную позицию списка
LoadFromFile	const FileName: string	Заполняет список строками текста из указанного текстового файла
Move	CurIndex, NewIndex: Integer	Перемещает строку в списке
SaveToFile	const FileName: string	Сохраняет строки списка в указанный файл
Sort	-	Производит сортировку строк списка

Наиболее часто используемым методом, безусловно, является Add, при помощи которого как раз и создаются списки. При необходимости вставить строку в какое-либо определенное место списка, вместо него используют Insert. Для управления отдельными элементами списков пригодятся Delete, Move и Exchange, а для операций над списком в целом – Clear, SaveToFile и LoadFromFile.

Пример, наглядно демонстрирующий работу целого ряда методов списков строк применительно к свойству Lines многострочного редактора можно посмотреть в каталоге Demo\Part3\StringList.

Что касается методов, предназначенных для поиска и сортировки, то к их использованию следует подходить с некоторой осторожностью. Прежде всего, время, необходимое для сортировки списков, возрастает вместе с его размерами не линейно, а в геометрической прогрессии. Соответственно, если сортировка списка из сотни другой элементов происходит почти моментально, то время выполнения сортировки для больших списков, содержащих многие тысячи строк, может оказаться весьма продолжительным. В то же время, если список является отсортированным изначально (т.е. свойство Sorted установлено в истину), то в процессе добавления новых строк они сразу же будут размещаться согласно порядку сортировки. Это следует учитывать при написании программ, работающих с объемными списками.

Списки

Помимо многострочного редактора, в Delphi, на той же закладке Standard палитры компонентов, располагаются еще 2 компонента, использующих списки строк для хранения своих значений. Это ListBox и ComboBox.

Простой список, представленный компонентом ListBox, представляет собой прямоугольную область, в которой располагаются его элементы – строки. Если строк в списке больше, чем может поместиться в отведенной области, то автоматически появляется полоса прокрутки.

Класс TListBox является наследником класса TWinControl и имеет собственные свойства, представленные в таблице 12.5.

Таблица 12.5. Основные свойства ListBox

Свойство	Тип	Описание
AutoComplete	Boolean	Определяет, должен ли список реагировать на нажатие клавиш таким образом, чтобы находить и выделить совпадающий элемент
BorderStyle	TBorderStyle	Определяет, должна или нет быть рамка вокруг списка. Допустимые значения: bsNone, bsSingle
Columns	Integer	Определяет количество колонок, видимых без горизонтальной прокрутки
Count	Integer	Указывает на количество элементов списка
ItemIndex	Integer	Определяет порядковый номер выбранного элемента, начиная с 0. Если не выбрано ни одного, то устанавливается в -1
Items	TStrings	Содержит строки списка
MultiSelect	Boolean	Определяет, допустимо или нет производить множественный выбор
SelCount	Integer	Указывает на количество выбранных элементов
Selected	array of Boolean	Определяет, выбран или нет тот или иной элемент списка
Sorted	Boolean	Определяет, должен ли список быть отсортирован

TopIndex	Integer	Определяет порядковый номер элемента, который является самым верхним в видимой части списка

Отдельное пояснение следует дать для свойства Columns. Дело в том, что если оно имеет значение, отличное от принятого по умолчанию нуля, то список становится не простым вертикальным, а многоколоночным, с горизонтальной прокруткой по колонкам для случая, если все элементы не умещаются на отведенной для списка области. Поэтому, в зависимости от того, какое значение имеет параметр Columns, один и тот же список может выглядеть совершенно по-разному (рис. 12.3).

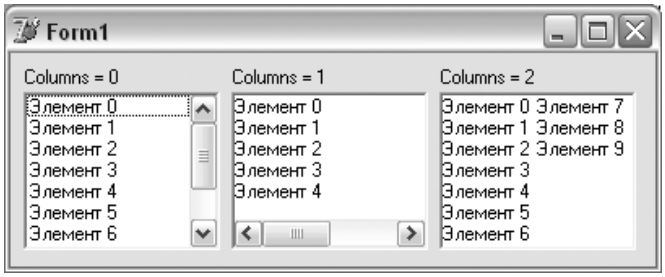

Рис. 12.3. Влияние значения свойства Columns на внешний вид списка

ПРИМЕЧАНИЕ

> Следует отметить, что многоколоночные списки используются довольно-таки редко. Пожалуй, едва ли не единственное их применение – это проводник Windows в режиме списка. При этом в нем используется совершенно другой компонент, аналогом которого в VCL является ListView.

Рассмотрим также свойство MultiSelect, которое оказывает непосредственное влияние на ряд других свойств. Так, если оно установлено в истину, то свойство ItemIndex всегда будет равно нулю, а если в ложь, то свойство SelCount всегда будет иметь значение -1. Что касается его непосредственного влияния на список, то оно позволяет выделять сразу несколько элементов с использованием мышки и клавиш Shift и Ctrl.

Если рассматривать методы списка, то основная их часть предназначена для выполнения тех или иных манипуляций над выбранными элементами:

- ClearSelection – снимает выделение со всех выбранных элементов;
- CopySelection – копирует выбранные элементы в другой список;
- DeleteSelected – удаляет все выделенные элементы из списка;
- SelectAll – выделяет все элементы в списке.

Еще 2 метода выполняют работу, характерную для списков вообще. Так, метод Clear удаляет все элементы списка, а метод AddItem добавляет новый элемент, и, при необходимости, ассоциирует с ним объект:

```
List1.AddItem('Строка - новый элемент списка',nil);
```

В данном случае вместо ссылки на объект был использован указатель nil, т.е. никакого объекта не было ассоциировано.

ПРИМЕЧАНИЕ

> Следует учитывать, что при работе с визуальными компонентами, представляющими списки, в случаях, когда одна и та же задача решается как методами свойства Items, так и собственными методами компонента, желательно использовать методы самих компонент. Т.е. List1.Clear предпочтительнее, чем List1.Items.Clear.

Теперь рассмотрим другой вариант списков, представленный компонентом ComboBox. Этот компонент расположен на палитре инструментов непосредственно за ListBox. Он интересен тем, что как бы объединяет в себе сразу 2 компонента – строку редактирования (Edit) и список (ListBox).

При работе с таким элементом интерфейса, называемым комбинированным, или ниспадающим списком, пользователь может как вводить собственный текст, так и выбирать один из предусмотренных в списке вариантов. Тем самым преодолевается такое ограничение последнего, как невозможность непосредственного ввода текста.

Очевидно, что ниспадающий список имеет ряд свойств, характерных как для однострочного редактора (SelText, SelStart, SelLength, MaxLength и CharCase), так и для обычного списка (Sorted, Items, ItemIndex и AutoComplete). Вместе с тем, предусмотрен и ряд уникальных свойств, в частности, необходимых для настройки его ниспадающей части. К ним относятся следующие свойства:

- AutoCloseUp – если установлено в истину, то раскрывающаяся часть списка будет закрываться автоматически, как только пользователь выберет элемент;
- AutoDropDown – если установлено в истину, то список будет раскрываться автоматически, как только пользователь начнет вводить текст;
- DropDownCount – это свойство определяет количество строк, которое будет отображаться в раскрывшемся списке;
- DroppedDown – обратившись к этому свойству, можно определить, раскрыт или нет список в данный момент.

Помимо всех этих свойств, ориентированных непосредственно на ниспадающую часть списка, имеется еще одно – Style, отвечающее за вид и поведение комбинированного списка. Среди возможных вариантов значений этого свойства нам интересны следующие:

- csDropDown – это значение принято по умолчанию, при нем список ведет себя именно так, как описано;
- csDropDownList – если установить это значение, то возможность непосредственного редактирования текста пропадет, т.е. подобно обычному списку пользователь сможет лишь выбирать из предложенных вариантов;
- csSimple – если оставить размеры списка по вертикали без изменений, то мы получим, фактически, элемент Edit, с «привязанным» к нему списком, который при этом не будет виден. Однако если размеры увеличить, растянув его по высоте, то непосредственно под ним появится ни что иное, как обычный список.

Чтобы нагляднее представить себе разницу между этими режимами, достаточно сделать простейшее приложение, использующее компонент ComboBox во всех возмож-

ных режимах, включая оба случая для csSimple – с видимой и скрытой частью списка. Пример можно найти в каталоге Demo\Part3\ComboBox.

Что касается методов комбинированного списка, то все они уже были нами рассмотрены в рамках обычного списка. Это AddItem, Clear, ClearSelection, CopySelection, DeleteSelected и SelectAll.

Переключатели

Переключатели являются еще одним видом основных, часто используемых элементов управления. Переключателями они называбтся потому, что могут находиться в 2 состояниях – в выбранном или невыбранном.

Переключатели бывают 2 типов – зависимыми (собственно переключатель, известный так же как «радиокнопка») и независимыми (флажки). Разница между ними заключается в том, что в группе зависимых переключателей может быть выбран только один, а независимые никак не связаны между собой. Визуально зависимые переключатели выглядят как кружки, а независимые – как квадратики. При щелчке по переключателю он меняет свое состояние, т.е. если был включенным (отмеченным), то становится выключенным, и наоборот.

Начнем с независимых переключателей – флажков. В VCL флажок представлен компонентом CheckBox. Он выглядит как небольшой прямоугольник с текстовым заголовком, расположенным справа. Если же вдруг появится острая необходимость «развернуть» флажок таким образом, чтобы текст находился слева, то можно установить свойство Alignment в taLeftJustify.

Но самым важным свойством флажка, пожалуй, является Checked. Именно оно определяет его состояние. Так, если флажок включен (что визуально проявляется как наличие галочки на квадратике), то это свойство имеет значение истины, а если выключен – лжи. Этим свойством можно управлять программно, визуально это будет проявляться как появление или исчезновение галочки.

Вместе с тем, для флажка предусмотрено еще одно состояние – запрещенное. Это состояние является опциональным и за его наличие или отсутствие отвечает свойство AllowGrayed. В том случае, если оно установлено в истину, то при щелчке мышкой по такому флажку будет происходить циклическая смена между 3 состояниями: установлен, снят и запрещен. В последнем режиме он выглядит как отмеченный, но с очень низкой контрастностью (рис. 12.4).

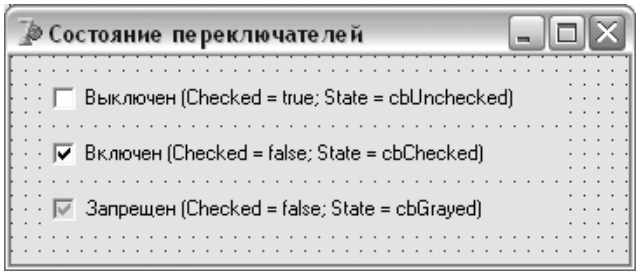

Рис. 12.4. Все состояния флажка

Поскольку состояний может быть не 2, а 3, то булевского значения в таких случаях оказывается недостаточно, чтобы представить все варианты. В таком случае для контроля или назначения состояния флажка используют свойство State. Оно может принимать следующие 3 значения типа TCheckBoxState:

- cbUnchecked – флажок не отмечен;
- cbChecked – флажок отмечен;
- cbGrayed – флажок недоступен.

Следует учитывать, что при изменении состояния переключателя не только пользователем (щелчком мышкой или нажатием пробела, когда элемент имеет фокус ввода), но и программно, происходит событие onClick. Это бывает удобным, когда требуется выполнять те или иные инструкции, в зависимости от текущего состояния флажка. Скажем, его включение или выключение может быть признаком доступности других элементов интерфейса на той же форме.

Второй вид переключателей – зависимые – представлен в Delphi компонентом RadioButton (отсюда и его жаргонное название «радиокнопка»). Как уже было отмечено, его основным отличием от флажка является то, что изменение состояния одного такого переключателя влияет на состояние других. Иначе говоря, только один зависимый переключатель в группе может быть отмечен. Поэтому такие переключатели обычно используют для указания взаимоисключающих опций. Соответственно, щелчком мышки его можно только включить. Для отключения же потребуется включить другой зависимый переключатель из этой же группы. Примерно так работают кнопки на старых радиоприемниках, что объясняет название этого компонента.

Поскольку состояний у такой кнопки может быть только 2 – включено или выключено, то и свойств остается только 2 – Checked и Alignment.

Рассмотрим поведение флажков и радиокнопок на примере. Для этого создадим новое приложение, на главную форму которого поместим 2 столбика переключателей: независимые – слева, а зависимые – справа. Для одного из флажков, скажем, последнего, установим свойство AllowGrayed в истину. Затем внизу формы расположим обычную кнопку, а для ее свойства Caption укажем «Сброс» (рис. 12.5).

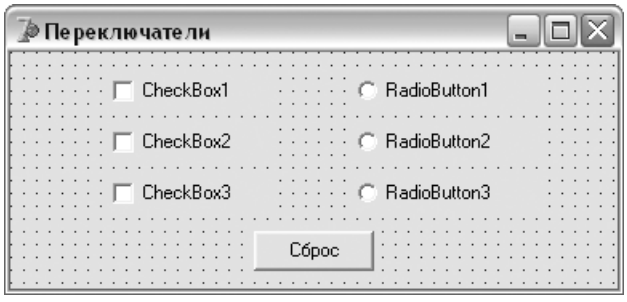

Рис. 12.5. Переключатели

Запустив приложение на выполнение, можно убедиться, что выставление флажков никак не влияет друг на друга, зато при щелчке по любой из радиокнопок только она становится выбранной. Отметим, что при проектировании формы мы не сделали выбранным ни один из переключателей. Соответственно, после того, как хотя бы один

раз был произведен щелчок по любой радиокнопке, вернуть все к начальному виду не удастся. Именно потому мы и предусмотрели кнопку сброса. Остается написать лишь код для нее.

Реализовать сброс всех флажков можно последовательно, банальным перечислением каждого элемента и выставления свойства Checked в ложь:

```
CheckBox1.Checked:=false;
CheckBox2.Checked:=false;
...
RadioButton3.Cgecked:=false;
```

В принципе, когда флажков всего несколько штук, этот вариант является оптимальным. Тем не менее, рассмотрим и другой способ, который может оказаться полезным при большом числе переключателей. Состоит он в обходе всех компонентов формы при помощи цикла по свойству Controls и в присваивании нужного значения нужному свойству всех подходящих компонентов. В нашем случае задача несколько усложняется, поскольку мы имеем 2 разных класса компонентов, и свойство Checked у них происходит от разных предков. Поэтому нам придется реализовать проверку на принадлежность объекта к тому или иному классу, и действовать уже исходя из этих данных. В результате мы получим следующий обработчик события для щелчка мышкой по кнопке:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i: integer;
begin
 for i:=0 to Form1.ControlCount-1 do begin
   if Form1.Controls[i] is TCheckBox then
     (Form1.Controls[i] as TCheckBox).Checked:=false;
   if Form1.Controls[i] is TRadioButton then
     (Form1.Controls[i] as TRadioButton).Checked:=false;
 end;
end;
```

Прежде всего, здесь организован цикл, обходящий все дочерние компоненты формы. В теле цикла сначала проверяется, принадлежит ли текущий компонент классу TCheckBox. Если да, то он явно приводится к типу TCheckBox при помощи оператора as, после чего с ним можно поступать так же, как если бы это было явное указание на объект этого класса, т.е. идет обращение к свойству Checked, которому присваивается новое значение. Затем то же самое проделывается для случая, когда компонент оказывается принадлежащим классу TRadioButton.

Такой подход, с использованием обязательной проверки на принадлежность к типу (с применением оператора is) исключает возможные ошибки времени выполнения. Например, если даже все переключатели были бы одного типа, скажем, флажками, нельзя упускать из виду того, что на форме кроме них могут находиться и другие компоненты. Даже в рассматриваемом нами случае такой компонент имеется - это та же кнопка «Сброс», которую, разумеется, не получилось бы привести от типа TButton к типу TCheckBox.

Работающий пример вы найдете в каталоге Demo\Part3\Checks.

Панели и группы

При рассмотрении радиокнопок неоднократно отмечалось, что они взаимодействуют в рамках группы. Очевидно, что в приведенных примерах группу образовавали все компоненты типа RadioButton, находящиеся на форме. Но это не означает, что на форме может быть только одна группа таких кнопок. Дело в том, что принадлежность компоненты к группе определяется на основании того, какому компоненту она принадлежит. Во всех рассмотренных ранее примерах таковым являлась сама форма. Но помимо формы имеются и другие компоненты, которые могут иметь дочерние объекты. Одним из наиболее широко применяемых элементов такого типа является панель – Panel. На палитре компонентов она расположена предпоследней на закладке Standard и выглядит как пустой серый квадратик. Поместив этот компонент на форму, вы получите прямоугольную область, ограниченную выпуклой рамкой.

Панели используются не только для логической группировки элементов управления, но и для визуального оформления приложений. Благодаря последнему обстоятельству, панель имеет целый ряд свойств, ответственных за внешний вид панели, вернее, ее рамки. Всего таковых имеется 5 штук, включая уже хорошо знакомо по другим элементам, например, текстовым редакторам, свойство BorderStyle. Точно так же, как и для них, для панели можно либо включить рамку (bsSingle), либо оставить ее выключенной (bsNone). Да, ошибки здесь нет: хотя рамка по умолчанию выключена, панель все равно имеет визуальные границы. А дело в том, что помимо собственно рамки, панель имеет еще и скосы (bevels), причем их 2 – внешние и внутренние. И именно внешний скос мы и видим при стандартных настройках панели. Соответственно, оставшиеся 4 свойства как раз скосами и управляют:

- BevelInner – это свойство отвечает за Сид и наличие внутреннего скоса;
- BevelOuter – это свойство управляет видом внешнего скоса;
- BevelWidth – при помощи этого свойства можно управлять толщиной линии, образующей скосы, причем данное свойство одновременно влияет как на внутренний, так и на внешний;
- BorderWidth – данное свойство позволяет изменять границу между внутренним и внешним скосами.

Для свойств, отвечающих за вид скосов – BevelInner и BevelOuter предусмотрено по несколько значений:

- bvNone – скос отсутствует (принято по умолчанию для внутреннего скоса, BevelInner);
- bvRaised – скос образует выпуклую рамку (принято по молчанию для внешнего скоса, BevelOuter);
- bvLowered – скос образует вогнутую рамку;
- bvSpace – в типичном случае аналогичен bvRaised.

Таким образом, мы получаем весьма широкие просторы для манипулирования внешним видом границ панели – можно как отключать границы панелей вообще, установив свойства BevelInner и BevelOuter в bvNone, так и заключать панели в объемные рамки самого разнообразного вида (рис. 12.6).

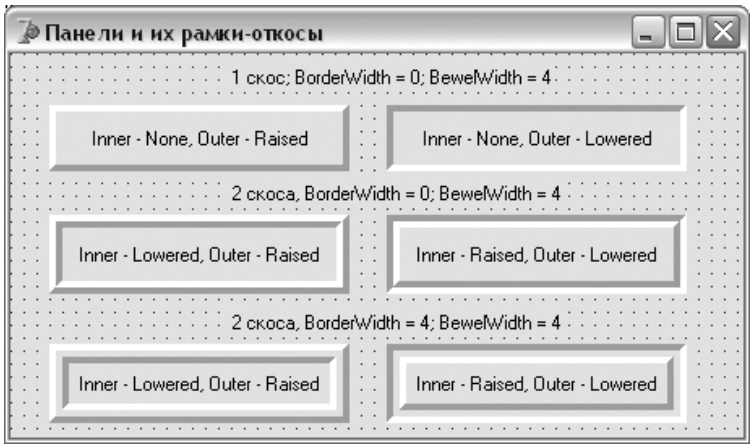

Рис. 12.6. Различные комбинации откосов панелей

Чтобы убедиться в том, что каждая панель образует группу компонент, попробуем использовать их для группировки радиокнопок. Для этого поместим на форму 2 панели, немного увеличим их размеры в высоту, и поместим на каждую по 2 компонента RadioButton. Если теперь запустить программу и попробовать изменить состояние переключателей на одной из панелей, то это никак не отразится на состоянии переключателей, расположенных на другой панели.

Следует отметить, что если панель используется не в декоративно-пояснительных целях (например, как подзаголовок окна), а именно как группирующий элемент, то ее свойству Caption присваивают пустую строку, а пояснительный текст размещают в верхнем правом углу при помощи компонента Label. При этом, как правило, скосы не используются (оба отвечающих за их вид свойства устанавливают в bvNone), а если необходимо визуально подчеркнуть, что размещенные на панели компоненты относятся к одной логической группе, используют обычную рамку (свойство BorderStyle).

Вместе с тем, в Delphi имеется компонент, специально предназначенный для группировки элементов – GroupBox, или контейнер группы. В отличие от панели, он не имеет откосов, но всегда заключен в рамку. При этом его заголовок (Caption) изначально расположен как раз там, где надо – в верхнем левом углу, причем накладывается поверх образующей рамку линии. В целом можно отметить, что класс TGroupBox происходит от TWinControl и не имеет каких-либо дополнительных свойств и методов.

Контейнер группы используется для объединения ряда различных компонент в один логически связанный блок. При этом компоненты могут быть использованы самые разные – переключатели, поля редактирования, кнопки и т.д. Если же в группе требуется разместить исключительно радиокнопки, то можно использовать другой компонент – RadioGroup. Его можно назвать контейнером группы зависимых переключателей. Этот специализированный компонент является наследником контейнера группы, предназначенным исключительно для радиокнопок. Причем непосредственно помещать на него компоненты RadioButton нет надобности, вместо этого следует использовать его собственное свойство Items. Каждый элемент, указанный в списке Items, является надписью с переключателю. Соответственно, задав, скажем, 4 строки в спи-

ске Items, мы получим готовую группу из 4 радиокнопок, являющуюся одним компонентом. Это очень удобно, поскольку вместо проверки состояния каждого из переключателей достаточно узнать свойство ItemIndex компонента RadioGroup. Для примера рассмотрим оба варианта: один – с обычной группой и 4 компонентами RadioButton, и другой – с компонентом RadioGroup и 4 строками, заданными в списке Items. В первом случае нам понадобится написать 4 строки кода:

```
if RadioButton1.Checked then Label1.Caption:='Вариант 1';
...
if RadioButton4.Checked then Label1.Caption:='Вариант 4';
```

Во втором же будет достаточно всего одной строки:

```
Label2.Caption:='Вариант '+IntToStr(RadioGroup1.ItemIndex+1);
```

Этот пример можно посмотреть в каталоге Demo\Part3\Groups.

Кроме свойств Items с надписями к переключателям, параллельно определяющим их количество, и ItemIndex, указывающего на выбранный элемент (если не выбран ни один вариант, то ItemIndex имеет значение -1), компонент RadioGroup имеет еще одно свойство – Columns. Оно отвечает за количество столбцов, по которым будет разделен список.

Меню

К настоящему моменту мы уже рассмотрели все визуальные компоненты, расположенные на закладке Standard палитры компонентов Delphi. Однако на ней имеются все еще не упомянутые компоненты, без которых невозможно представить ни одного сколько-нибудь серьезного приложения. Это – меню.

Всего существует 2 типа меню – главное меню – то, что располагается непосредственно под заголовком окна программы, и локальное, всплывающее или контекстное меню, привязанное к тому или иному элементу управления. И если главное меню предназначено для управления работой всего приложения, то каждое контекстное меню служит для управления каким-либо конкретным элементом интерфейса.

Для создания главного меню в VCL предусмотрен компонент MainMenu, а для контекстного – PopupMenu. Оба они происходят от класса TMenu, и имеют некоторые общие свойства. В частности, свойство AutoHotkeys определяет, должны ли автоматически назначаться символы для быстрой навигации по меню (вы видите их подчеркнутыми при нажатой клавише Alt). А свойство AutoLineReduction определяет, должны ли автоматически удаляться повторяющиеся строки-разделители. Оба эти свойства могут принимать значения maAutomatic и maManual.

Так же у них присутствуют свойства Images, позволяющие привязать к меню коллекцию изображений, и Items, являющиеся хранилищем самих пунктов меню.

Что касается индивидуальных свойств этих компонентов, то здесь для нас наибольший интерес представляют настройки контекстного меню. Так, свойство Alignment определяет, в какой позиции относительно указателя мышки должно появиться меню. Допускается выравнивание его левого верхнего угла по левому краю (paLeft, принято по умолчанию), по правому (paRight) и по центру (paCenter). Другое свойство, AutoPopup, Отвечает за появление контекстного меню на экране. Если оно установлено в истину, то меню будет появляться автоматически при щелчке правой кла-

вишей мышки по компоненту, к которому это меню привязано. Если же установить значение этого свойство в ложь, то процесс появления меню должен будет контролироваться программно. Отметим, что для показа контекстного меню таким способом следует использовать метод Popup. Например, чтобы привязать всплытие меню к любому щелчку мышки по форме, можно написать примерно такой обработчик события для onClick:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 PopupMenu1.Popup(X, Y);
end;
```

Единственным спорным моментом в данном случае будет место появления меню. Дело в том, что аргументами X и Y, передаваемыми обработчику события onClick формы будут локальные координаты, отсчитываемые от верхнего левого угла ее окна. В то же время для метода Popup необходимы координаты относительно всего экрана (т.е. как для объекта Screen). Выходом из данной ситуации будет использование информации о координатах самой формы, которые так же привязаны к глобальной системе координат:

```
PopupMenu1.Popup(Form1.Left+X,Form1.Top+Y);
```

В принципе, всех уже перечисленных свойств, вкупе с методом Popup, вполне достаточно для того, чтобы приступить к использованию меню. Однако не следует забывать, что состав меню определяется его свойством Items – коллекцией пунктов меню. При этом каждый пункт представляет собой объект класса TMenuItem. Именно его свойства мы и рассмотрим, для чего обратимся к таблице 12.6.

Таблица 12.6. Свойства MenuItem

Свойство	Тип	Описание
Action	TBasicAction	Ссылается на действие, связанное с данным пунктом меню. Все остальные свойства этого пункта меню в таком случае задаются косвенно, посредством компоненты ActionList
AutoCheck	Boolean	Определяет, должен ли данный пункт менять свое свойство Checked при его выборе пользователем
AutoHotkeys	TMenuItemAutoFlag	Определяет, должны ли автоматически назначаться символы для быстрой навигации по подменю.
AutoLineReduction	TMenuItemAutoFlag	Определяет, должны ли автоматически удаляться повторяющиеся строки-разделители
Bitmap	TBitmap	Позволяет назначить индивидуальное графическое изображение данному пункту меню
Break	TMenuBreak	Определяет, является ли данный пункт началом нового столбца меню. Может принимать значения mbNone, nbBreak и mbBreakBar
Caption	String	Определяет заголовок пункта меню. Если в качестве заголовка использовать символ "-", то на месте такого пункта будет отображаться разделительная линия

Checked	Boolean	Определяет, является ли данный пункт меню отмеченным. Если да, то он выделяется соответствующей меткой
Count	Integer	Указывает на количество пунктов подменю в данном пункте меню. Нулевое значение означает, что подменю нет
Default	Boolean	Определяет, является ли данный пункт выбранным по умолчанию. Такие пункты выделяются полужирным шрифтом и автоматически вызываются при двойном щелчке по родительскому пункту меню
Enabled	Boolean	Определяет, является ли данный пункт доступным
ImageIndex	Integer	Указывает на порядковый номер изображения в коллекции (компонент ImageList), связанного с данным меню
Items	TMenuItems	Определяет массив пунктов вложенного подменю
RadioItem	Boolean	Определяет, является ли данный пункт составной частью взаимоисключающих пунктов меню
ShortCut	TShortCut	Определяет сочетание горячих клавиш для быстрого вызова данного пункта
Visible	Boolean	Определяет, должен ли данный пункт быть видимым пользователю

Очевидно, что поскольку любой пункт меню может содержать собственное подменю, то он должен иметь и свойства, характерные для основного компонента меню. Поэтому мы здесь снова видим AutoHotkeys и AutoLineReduction, которые, в дополнение к значениям maAutomatic и maManual могут также принимать значение maParent, что делает возможным наследование значения этих свойств от основного меню. Возможностью иметь дочерние подменю объясняются и такие свойства, как Count и Items. Что касается всех остальных свойств, то они предназначены для управления видом и поведением данного пункта меню.

Наиболее востребованными свойствами являются Caption и ShortCut. В большинстве случаев вам придется непосредственно работать только с этими двумя свойствами, лишь иногда прибегая к таким свойствам, как, скажем, Checked или ImageIndex. В то же время, некоторые свойства, например такие, как Visible или Enabled, редко определяются в процессе создания приложения. Для этих целей необходимо использовать соответствующий программный код, который будет изменять их состояние в зависимости от текущей ситуации. В качестве типичного примера можно привести функцию вставки из буфера обмена: если в нем ничего нет, то многие программы автоматически соответствующий пункт меню его недоступным. Фактически, для этого свойство Enabled устанавливается в значение «ложь».

Вообще же для создания меню в Delphi IDE предусмотрен специальный редактор, позволяющий наглядно создавать и редактировать как главное меню, так и контекстные. Для того, чтобы им воспользоваться, достаточно поместить нужный компонент (MainMenu или PopupMenu) на форму, после чего воспользоваться его собственным

контекстным меню. Первым пунктом будет Menu Designer, и как раз он и вызывает нужный редактор, называемый конструктором меню (рис. 12.7).

СОВЕТ

Подобно обычным визуальным компонентам вроде кнопки или поля редактирования, компоненты, подобные MainMenu, так же могут быть настроены на «действие по умолчанию», происходящее при двойном щелчке мышки. Для рассматриваемых компонентов таким действием как раз будет вызов конструктора меню.

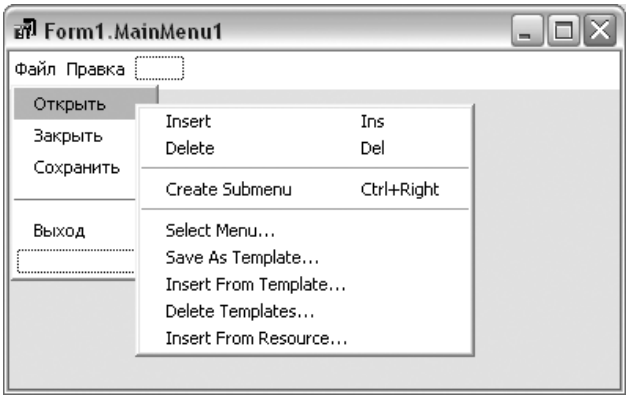

Рис. 12.7. Конструктор меню с раскрытым собственным контекстным меню

При помощи этого конструктора можно создавать меню, причем его внешний вид будет практически соответствовать тому, что получится на самом деле. Если в процессе работы щелкнуть правой кнопкой мышки по созданному пункту меню, то откроется контекстное меню самого редактора, при помощи которого этот пункт можно удалить (delete), либо добавить (Insert) вслед за ним новый, или же сделать данный пункт родительским для собственного подменю (Create Submenu). При этом свойства выбранного в данный момент пункта меню, видны и доступны для правки в инспекторе объекта.

Для того чтобы связать элемент меню с выполняемой им функцией, используется обработчик события onClick. Фактически, это единственное событие, которое заслуживает внимание. Причем оно происходит вне зависимости от того, что именно сделал пользователь для выбора данного пункта: это может быть и щелчок мышкой, и выбор пункта при помощи навигации с клавиатуры с нажатием пробела, или же обращение к нему при помощи заданной комбинации горячих клавиш.

Что касается собственно комбинаций горячих клавиш, которые можно назначить меню, то они перечислены в окне инспектора объекта, когда вы раскрываете список напротив свойства ShortCut. При их назначении следует следить за тем, чтобы они не повторялись для разных пунктов.

Отдельно следует сказать о свойстве Action (действие). Устанавливать это свойство можно только в том случае, если на форму уже помещен компонент ActionList, представляющий собой упорядоченный набор действий. В таком случае перечень доступных вариантов определяется исключительно им. Кроме того, при этом ряд других свойств задается автоматически аналогичными свойствами выбранного действия.

Коллекция действий

При описании компонент, реализующих меню, мы уже сталкивались с некими действиями (actions). Действия объединяются в списки при помощи компонента ActionList, в котором все они будут храниться.

ПРИМЕЧАНИЕ

> Компонент ActionList был введен в Delphi 6 для обеспечения межплатформенной совместимости. В то же время, в Delphi имеется и другой компонент аналогичного назначения, предоставляющего еще больше возможностей – ActionManager.

Списки действий обычно используют для того, чтобы определить реакцию программы на те или иные действия пользователя. С одной стороны, это несколько дольше, чем просто присвоить какой-либо кнопке или пункту меню действие для, скажем, обработки события onClick. Но с другой, это позволяет хранить все действия централизованно, что, в конечном счете, упрощает процесс разработки приложения тем, что если действия для какого-то пункта меню и для другого элемента (скажем, кнопки на панели инструментов) совпадают, то им обоим можно присвоить одно действие. Кроме того, для целого ряда стандартных операций вроде работы с буфером обмена, открытия или сохранения файла, а так же для стандартных действий с окнами MDI-приложений, предусмотрены уже готовые действия.

Все операции по созданию и настройке действиями производятся в специальном окне компонента ActionList (рис. 12.8), которое вызывается либо из контекстного меню этого компонента (пункт Action List Editor), либо двойным щелчком мышки – в точности подобно тому, как это реализовано для компонент-меню.

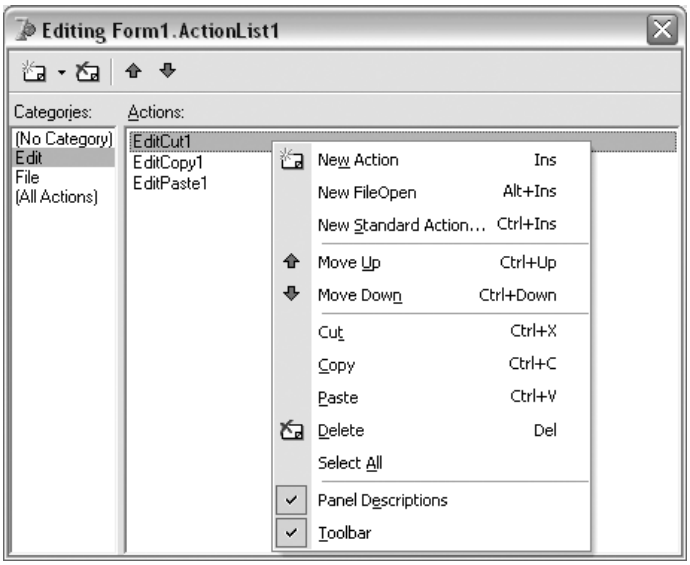

Рис. 12.8. Окно редактора списка действий и его контекстное меню

Добавляются новые действия либо при помощи контекстного меню, либо путем нажатия на кнопку-список New Action. Причем, если нажать не на саму кнопку, а не

стрелку рядом с ней и выбрать вариант New Standard Action (или выбрать аналогичный пункт из контекстного меню), то откроется другое окно, содержащее список с уже упомянутыми стандартными действиями.

Все стандартные действия подразделяются на 12 групп, каждая из которых содержит ряд действий, принадлежащих к соответствующей категории:

- Edit (правка) – группа для стандартных действий, связанных с редактированием (копировать, вставить, выделить и т.д.);
- Format (формат) – группа действий, предназначенных для работы с форматированным текстом;
- Help (помощь) – типичные действия, помещаемые в пункт справка;
- Window (окно) – стандартные действия с окнами для MDI-приложений;
- File (файл) – обращение к диалогам открытия и сохранения файла, печать;
- Search (поиск) – действия, связанные с поиском и заменой в тексте;
- Tab (закладки) – действия для навигации между закладками для соответствующих элементов интерфейса;
- List (список) – ряд операций, применимых к элементам управления типа списков;
- Dialog – обращения к различным диалогам, кроме тех, что используются в группе File;
- Internet – некоторые действия, связанные с обращением к функциям Интернета (например, открыть адрес в браузере);
- Dataset – ряд действий, характерных для компонент, предназначенных для работы с базами данных (переход между записями, запросы и т.д.);
- Tools – эта группа состоит из единственного представителя, предназначенного для обращения к настройкам самих действий во время выполнения. Актуально только в том случае, если используется ActionManager.

Большинство стандартных действий вызывают различные диалоги, или служат для выполнения иных операций, нами еще не рассмотренных – с ними мы ознакомимся в следующих 2 главах. Тем не менее, достаточно добавить несколько действий, после чего почти что любому компоненту можно будет назначать в качестве значения для свойства Action действие. С этой точки зрения действия являются наиболее востребованными для того же главного меню.

В то же время, их вряд ли целесообразно использовать для диалоговых окон, равно как и в тех случаях, если сама по себе программа достаточно компактна, либо если она не является текстовым редактором. Кроме того, не следует забывать, что если вы не переопределили сочетания горячих клавиш вроде Ctrl+C и Ctrl+V, отвечающих за работу с буфером обмена, то они все равно будут работать в компонентах, предназначенных для работы с текстом.

То же самое можно сказать и про контекстное меню: если в программе не предусмотрено специфических контекстных меню для компонент типа Edit или Memo, то оно у них все равно будет – об этом позаботится операционная система.

Компоненты 32-разрядного интерфейса

Мы уже рассмотрели целый ряд компонент, являющихся основополагающими при разработке программ. Все эти компоненты объединяет то, что они используются на всех графических платформах, начиная с Windows 3.x и OS/2 и заканчивая Windows XP или Linux. Однако современные (и не очень) 32-разрядные версии Windows, позволяют использовать ряд дополнительных элементов пользовательского интерфейса, являющихся стандартными для всех версий Windwos, начиная с Windows 95. В этой главе мы рассмотрим ряд компонент, представляющих данные элементы интерфейса, включая такие, как строка состояния и панель инструментов.

Коллекция картинок

Для использования в приложении различных пиктограмм, поясняющих назначение пунктов меню, а также для оформления кнопок, расположенных на панелях инструментов, удобнее всего использовать не отдельные изображения, привязанные к каждому конкретному объекту, а коллекции. В таком случае компонент будет лишь ссылаться на номер картинки в такой коллекции, представленной в VCL компонентом ImageList.

Компонент ImageList, или список изображений, расположенный на закладке Win32 палитры компонентов, является невизуальным компонентом. Иначе говоря, на экране его содержимое не отображается непосредственно, но может быть выведено на поверхности какого-либо визуального компонента. При этом элементами ImageList могут выступать изображения форматов «битовый массив» bmp или «значок» ico.

Размеры всех картинок в списке должны быть одинаковыми. Чтобы задать размер каждого изображения, используют свойства Height и Width, определяющие габариты картинки в пикселях.

СОВЕТ

> Подборку, включающую свыше 1000 картинок, которые можно использовать при разработке приложений, вы найдете в каталоге Clipart на прилагаемом к этой книге компакт-диске.

Кроме самих изображений, в списке могут храниться маски изображений, определяющие способ вывода самого изображения. Размеры маски должны совпадать с размерами самого изображения. Так, нулевой бит маски означает, что в данной точке изображение должно быть прозрачным (т.е. будет выведен пиксель цвета фона). Ненулевой бит указывает на то, что в данном месте должен быть выведен соответствующий ему бит изображения. За возможность обработки маски изображения отвечает свойство Masked. Если оно установлено в истину, то маска будет накладываться при выводе, а если в ложь, маска учитываться не будет.

Если при выводе изображения требуется заменить цвет фона в тех местах, где оно является прозрачным, следует использовать свойство bkColor. При значении, отличном от clNone («бесцветный»), указанный в этом свойстве цвет будет использоваться

в качестве фона на области, занимаемой изображением. Однако если свойство Masked установлено в ложь, то значение свойства bkColor не будет иметь смысла.

Если же маска как таковая не указана, то для определения прозрачной области используется цвет фона. Чтобы его указать, да и для того, чтобы вообще определить сами изображения во время разработки приложения, используют специальный редактор (рис. 13.1), который можно вызвать из контекстного меню компонента (ImageList Editor), или двойным щелчком по нему.

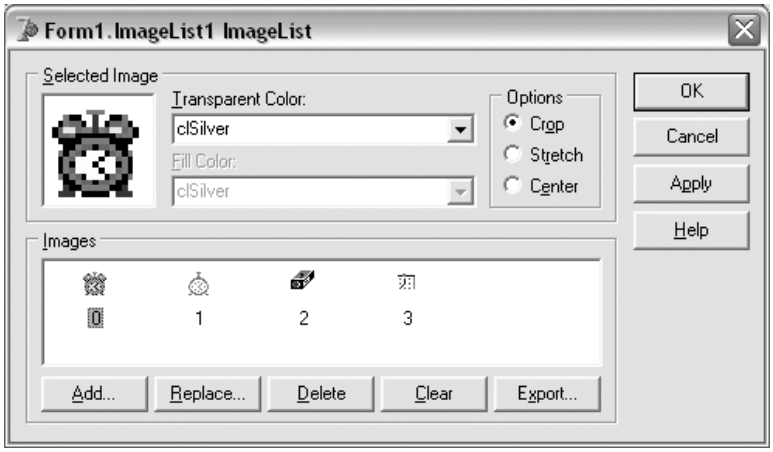

Рис. 13.1. Редактор ImageList

При помощи редактора ImageList можно вносить новые изображения в список, удалять их, менять местами и т.д. Непосредственно после внесения можно так же задать цвет прозрачности (Transparent color), т.е. тот, который будет при выводе заменен цветом фона. Этот цвет можно выбрать из соответствующего списка, либо использовать инструмент-пипетку, доступный в области просмотра выбранного изображения (Selected Image).

Для внесения изображения в список при помощи редактора используйте кнопку Add, для замены выбранного изображения другим – Replace, а для удаления – Delete. Кнопка Clear удаляет все изображения из коллекции, а при помощи Export вы можете сохранить их все в виде одного файла. Здесь следует заметить, что сам компонент ImageList хранит все свои изображения в виде общего образа, и при экспорте именно этот образ и будет сохранен как файл bmp.

Что касается использования ImageList, то любой компонент, имеющий свойство Images (например, Menu или PopupMenu), можно ассоциировать со списком изображений, указав для данного свойства имя доступного для данной формы объекта ImageList. После этого для ссылки на изображение достаточно будет указывать его порядковый номер в списке. Он задается при помощи свойства ImageIndex, имеющегося, например, у всех пунктов меню – MenuItem.

Индикатор выполнения

Для индикации выполнения продолжительных операций – таких, как процесс копирования множества файлов, форматирования дисков и т.д., используют специальный

графический индикатор, обычно представляющий собой полосу с дискретным приращением. Индикатор выполнения представлен в VCL компонентом ProgressBar, и, подобно другим рассматриваемым в этой главе компонентам, находится на закладке Win32 палитры компонентов.

Компонент ProgressBar является наследником класса TWinControl и имеет всего несколько свойств, необходимых для своего использования по назначению, а именно:

- Min и Max – минимальное и максимальное значения диапазона индикатора. Задаются целочисленным значением, по умолчанию минимальное значение равно 0, а максимальное – 100;
- Position – позиция, или текущее значение индикатора. Значение этого параметра всегда должно находиться в границах диапазона, заданного свойствами Min и Max;
- Step – величина приращения при дискретном изменении значения методом StepIt. По умолчанию установлено в значение 10;

Кроме этих свойств, необходимых для непосредственного использования индикатора выполнения, так же предусмотрены свойства, влияющие на его отображение. Прежде всего, это свойство Orientation, которое влияет на пространственную ориентацию индикатора. Оно может принимать 2 значения – pbHorizontal и pbVertical, означающие размещение по горизонтали и вертикали. По умолчанию используется горизонтальное размещение, если же потребуется изменить его на вертикальное, то следует соответствующим образом изменить размеры (ширину и высоту) компонента.

Другое свойство – Smooth булевого типа, может пригодиться в том случае, если вопреки рекомендациям Microsoft по части интерфейса Windows-приложений, вы решитесь использовать не дискретную полосу индикации, а сплошную. В таком случае достаточно будет установить значение свойства Smooth в истину.

Изменять значение индикатора можно несколькими способами. Прежде всего, можно устанавливать значение непосредственно через свойство Position:

```
ProgressBar1.Position:=50;
```

Другой способ состоит в использовании уже упомянутого метода StepIt. При этом значение индикатора будет увеличено на величину, указанную в свойстве Step. Это удобно в тех случаях, когда, например, требуется выполнить заранее известное число операций (скажем, 5), примерно одинаковых по времени выполнения. В таком случае можно, оставив значения Min и Max принятыми по умолчанию, установить параметр Step в 20, и использовать метод StepIt для отображения хода выполнения:

```
ProgressBar1.Step:=20;
for i:=1 to 5 do begin
  RunLongProc(i);
  ProgressBar1.StepIt;
end;
```

В тех же случаях, когда требуется изменять значения индикатора на разные величины, то, помимо прямого указания нового положения при помощи свойства Position, можно использовать метод StepBy, передавая ему в качестве параметра значение, на которое должно увеличиться значение индикатора.

Ползунок

Если индикатор выполнения призван отображать какие-либо процессы, происходящие в программе, и пользователь не может непосредственно изменить его состояние, то компонент регулировки диапазона, или ползунок, решает полностью противоположную задачу, а именно – позволяет пользователю самому установить значение из диапазона.

Ползунок представлен компонентом TrackBar, и расположен непосредственно по соседству с компонентом ProgressBar. Это соседство не случайно, ведь хотя задачи этих компонентов и противоположны, по своей сути они оба являются компонентами, работающими с диапазонами значений. В частности, как и ProgressBar, компонент TrackBar является прямым наследником класса TWinControl. Кроме того, он так же имеет свойства Min, Max и Position. Единственным отличием пока является то, что по умолчанию для свойства Max установлено значение 10. Еще одно общее свойство – Orientation – также позволяет изменить расположение ползунка с горизонтального на вертикальное. Правда, при этом его размеры меняются автоматически.

Зато далее начинаются более существенные отличия. В частности, компонент TrackBar имеет целый ряд свойств, влияющих на его вид, в основном – на расположение его рисок. Все оставшиеся свойства этого компонента приведены в таблице 13.1.

Таблица 13.1. Свойства компонента TrackBar

Свойство	Тип значений	Описание
Frequency	Integer	Определяет величину интервала между рисками
LineSize	Integer	Определяет величину приращения при управлении ползунком с клавиатуры клавишами-стрелками
PageSize	Integer	Определяет величину приращения при управлении ползунком с клавиатуры клавишами PageUp и PageDown
SelEnd	Integer	Определяет позицию окончания выбранного диапазона
SelStart	Integer	Определяет позицию начала выбранного диапазона
SliderVisible	Boolean	Определяет, должен ли отображаться сам ползунок
ThumbLength	Integer	Определяет размер ползунка
TickMarks	TTickMark	Определяет расположение рисок. Допустимые значения: tmBottomRight, tmTopLeft и tmBoth
TickStyle	TTickStyle	Определяет способ вывода рисок. Допустимые значения: tsNone, tsAuto и tsManual

Итак, из приведенных в таблице 13.1 свойств, 3 отвечают за вывод рисок. Это Frequency, TickMarks и TickStyle. По умолчанию риски рисуются для каждого шага приращения (Frequency:=1), по нижней стороне полосы (TickMarks:=tmBottomRight). То, что они вообще выводятся – «заслуга» свойства TickStyle, имеющего по умолчанию значение tsAuto. Если бы оно было установлено в tsNone, то рисок вообще не было бы видно. В том же случае, если для него выбрано значение tsManual, то выделено было бы только начало и конец диапазона, а промежуточные позиции можно было бы выставить при помощи метода SetTick:

```
TrackBar1.TickStyle:=tsManual;
TrackBar1.SetTick(4);
TrackBar1.SetTick(7);
```

В данном случае у компонента TrackBar1 было бы выставлено 4 метки – в начале, на 3-й позиции, на 7-й, и на последней. Метод SetTick – это единственный собственный метод компонента TrackBar, все же остальные его методы вполне стандартны, поскольку являются унаследованными от класса TWinControl.

Еще 2 свойства – SliderVisible и ThumbLength отвечают за сам ползунок. Так, если свойство SliderVisible установлено в ложь, то ползунка видно не будет. При этом, разумеется, пользователь ничего не сможет поделать с таким компонентом. А свойства LineSize и PageSize влияют на перемещение ползунка при помощи клавиатуры.

Таким образом, остается рассмотреть еще 2 свойства – SelStart и SelEnd, которые, в дополнение к свойству Position, определяют некие выбранные из диапазона значения, фактически, образуя вложенный диапазон. Он выделяется на шкале ползунка серым цветом. Отметим, что данные свойства можно установить только в процессе визуальной разработки или программно.

Для примера рассмотрим приложение, использующее 2 ползунка: один – для выбора значения, а другой – для установки коэффициента. Итоговое значение будет отображаться численно в заголовке окна и визуально – при помощи индикатора выполнения. Для этого создадим новое приложение и расположим на его главной форме 2 элемента TrackBar и 1 ProgressBar. Затем произведем следующие изменения ри помощи инспектора объекта:

- Для компонента TrackBar1 установим свойство Orientation в trVertical, а свойствам Min, Max и Position назначим значения 6, 12 и 10, соответственно. Кроме того, установим для него свойства SelEnd и SelStart в 11 и 9, а TickMarks – в tmBoth;
- Для TrackBar2 ограничимся изменением значения его свойства ThumbLength с 20 на 15;
- Индикатор ProgressBar1 оставим вообще без изменений.

После всех проделанных операций разместим все эти компоненты на форме таким образом, чтобы TrackBar1 находился по левому краю формы, а оставшиеся 2 компонента – правее его. Кроме того, снабдим их все поясняющими надписями при помощи меток: для TrackBar1 – «коэффициент», для TrackBar2 – «перемещение», и для ProgressBar1 – «индикация» (рис. 13.2).

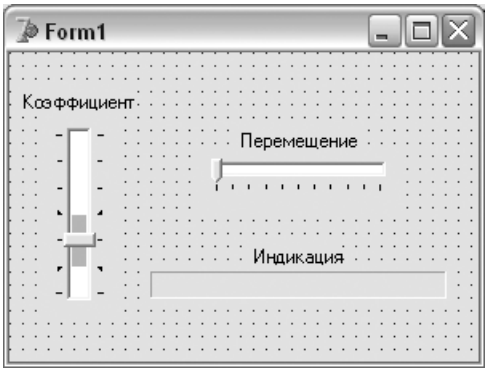

Рис. 13.2. Форма с 2 ползунками и 1 индикатором

Идея нашего приложения состоит в том, чтобы пользователь, изменяя положение ползунка, помеченного как «перемещение» (TrackBar2), мог бы влиять на положение индикатора (ProgressBar1). При этом величина перемещения задается при помощи коэффициента (TrackBar1). Поскольку индикатор имеет максимальное значение 100, а количество позиций перемещения – 10, то для полного совпадения в конечной точке понадобится коэффициент 10 – для этого и было сделано выделение диапазона от 9 до 11.

Весь программный код данного приложения сводится к обработчику события onChange элемента TrackBar2. Чтобы создать такой обработчик, достаточно дважды щелкнуть по размещенному на форме объекту TrackBar2. Сам же код будет состоять из вычисления позиции для индикатора и вывода его числового значения в заголовке формы:

```
procedure TForm1.TrackBar2Change(Sender: TObject);
begin
 ProgressBar1.Position:=TrackBar2.Position*TrackBar1.Position;
 Form1.Caption:=IntToStr(ProgressBar1.Position);
end;
```

Результат проделанной работы можно найти в папке Demo\Part3\TrackProg.

Реверсивный счетчик UpDown

Еще один компонент, предназначенный для работы с диапазонами значений – это UpDown, известный так же как реверсивный счетчик. Внешне он представляет собой сдвоенную кнопку с двумя стрелками, используемыми для увеличения или уменьшения значения. Поскольку сам по себе этот компонент ничего не отображает, то его обычно ассоциируют с каким-либо другим оконным объектом, который может визуально отображать значение счетчика. Как правило, для этих целей используют однострочный редактор – Edit, метку типа StaticText (Label не подходит по той причине, что не является оконным компонентом), или любой другой компонент, который может выводить текст (многострочный редактор, ниспадающий список и т.д.).

Подключение счетчика к ассоциированному компоненту производится путем присваивания свойству Associate компонента UpDown имени нужного объекта. Для этого можно использовать инспектор объекта: в раскрывающемся списке напротив этого свойства будут перечислены все подходящие для этого объекты. При ассоциировании счетчик автоматически выравнивается по правой стороне выбранного объекта. Но при помощи свойства AlignButton можно сделать так, чтобы счетчик оказался по правой стороне компонента – для этого данное свойство надо установить в значение udLeft.

Еще одно свойство компонента UpDown – Orientation – уже знакомо нам по другим компонентам, предназначенным для работы с диапазонами значений. Применительно к реверсивному счетчику оно может принимать значения udVertical и udHorizontal, и меняет расположение кнопок между видами вверх-вниз и влево-вправо.

Другие свойства также нам уже известны – это Min, Max и Position. Свойство же Increment по своей сути аналогично свойству LineSize у компонента TrackBar: оно отвечает на изменение значения при нажатии пользователем на стрелки клавиатуры или при щелчке мышкой по кнопке. Правда, для реверсивного счетчика имеются не-

которые отличия: во-первых, он реагирует только на нажатие стрелок вверх и вниз, а во-вторых, его реакцию на клавиатуру можно вообще отключить, если установить в ложь другое его свойство – ArrowKeys.

Наконец, последнее свойство, Thousands, отвечает за то, должны или нет использоваться разделители тысяч.

Поскольку реверсивные счетчики, чаще всего, используется совместно с однострочным редактором, то следует учитывать некоторые особенности их совместного использования. Прежде всего, пользователь может изменить текущее значение счетчика (т.е. его свойства Position) не только щелкая по кнопкам, но и введя его непосредственно в ассоциированный компонент с клавиатуры. При этом автоматически производится преобразование строки в число, и результат присваивается свойству Position. Очевидно, что при изменении значения самим компонентом UpDown, выполняется обратная ситуация: значение свойства Position преобразуется в строку и присваивается свойству Text редактора. Этим можно пользоваться для того, чтобы избегать лишних преобразований в коде программы, поскольку когда требуется получит число, то можно обратиться к свойству Position, а когда текст – к Text. Например, для элемента UpDown1, ассоциированного с Edit1, мы можем использовать такой код:

```
Form1.Caption:=Edit1.Text;
ProgressBar1.Position:=UpDown1.Position;
```

При этом следует учитывать, что если пользователь введет вместо числа строку, которую невозможно преобразовать в целое, то значение свойства Position не изменится, хотя исключительной ситуации при этом не возникнет.

Вместе с тем, в VCL предусмотрен специальный компонент, представляющий собой реверсивный счетчик, совмещенный с текстовым редактором. Он называется SpinEdit и расположен на закладке Samples палитры компонентов. Этот компонент не является стандартным для Windows, однако удобен тем, что не позволяет пользователю вводить какие-либо символы кроме цифр. Текущее значение этого компонента доступно только в числовом виде и представлено свойством Value. Другие свойства – MinValue и MaxValue аналогичны свойства Min и Max компонента UpDown, а свойство MaxLength, отвечающее за максимальное количество цифр, позаимствовано у компонента Edit. Ну а свойство Increment полностью соответствует таковому у счетчика UpDown.

Единственное новое свойство компонента SpinEdit – это EditorEnabled, которое позволяет запретить ввод данных непосредственно с клавиатуры. Впрочем, здесь также можно провести аналогию с парой UpDown-Edit, поскольку у однострочного редактора имеется свойство Enabled, позволяющее добиться такого же эффекта.

Редактор горячих клавиш

В VCL предусмотрены специальные средства для ввода не только диапазонов, но и других специализированных данных. Например, чтобы предоставить пользователю возможность указать сочетание горячих клавиш путем непосредственного их нажатия, в VCL предусмотрен специальный компонент – HotKey. Внешне он напоминает однострочный редактор, однако на этом, в общем-то, все сходство между этими компонентами заканчивается. Прежде всего, в качестве текста, отображаемого в окне редактора горячих клавиш, может выступать лишь текстовое представление одного

из допустимых сочетаний. Соответственно, свойством, представляющим текущее значение этого компонента, является не Text, а HotKey.

ПРИМЕЧАНИЕ

> Здесь следует пояснить, что сочетание клавиш, на самом деле, представляет собой тип TShortCut, являющийся диапазоном от Low(Word) до High(Word), т.е., фактически, положительным 16-битным целым. При этом каждому сочетанию соответствует то или иной числовое значение.

Кроме свойства HotKey, у компонента HotKey имеется еще 2 свойства, определяющих режим его работы и поддерживаемые сочетания клавиш. Так, свойство InvalidKeys позволяет ограничить набор допустимых сочетаний. Оно представляет собой множество типа THKInvalidKey, и может включать в себя следующие значения:

- hcNone – сочетания без модификаторов (Ctrl, Alt) недопустимы – принято по умолчанию;
- hcShift – клавиша Shift не может выступать модификатором, так же принято по умолчанию (т.е., скажем, сочетание Shift+A не может быть выбрано);
- hcCtrl – клавиша Control не может выступать модификатором;
- hcAlt – клавиша Alt не может быть модификатором;
- hcShiftCtrl – сочетание Shift и Control не может быть модификатором;
- hcShiftAlt – сочетание Shift и Alt не может быть модификатором;
- hcCtrlAlt – сочетание Control и Alt не может быть модификатором;
- hcShiftCtrlAlt – сочетание Shift, Control и Alt не может быть модификатором.

При этом следует учитывать, что если, скажем, запретить в качестве модификатора сочетание Ctrl+Alt и Shift+Ctrl то это не значит, что нельзя будет выбирать сочетания Shift+Ctrl+Alt.

Помимо запретов можно устанавливать и автоматически назначаемые модификаторы. Они применяются как значения по умолчанию. Например, в том случае, если введенный пользователем модификатор недопустим, то он будет заменен модификатором, заданном в качестве шаблона при помощи свойства Modifiers. Подобно свойству InvalidKeys, свойство Modifier представляет собой множество, состоящее из 4 значений: hkShift, hkCtrl, hkAlt и hkExt. По умолчанию используется hkAlt, т.е. неподходящий модификатор будет автоматически заменяться на Alt.

Компоненты для работы с датами

Другими специализированными компонентами, предназначенными для правки определенных типов значений, являются компоненты MonthCalendar и DateTimePicker. Оба они предназначены для упрощения ввода пользователем дат, попутно избавляя разработчика от заботы над тем, каким образом и в каком формате пользователь будет пытаться вводить даты.

Компонент MonthCalendar представляет собой календарь на 1 месяц: вверху находится поле, отображающее год и месяц в окружении 2 кнопок, позволяющих последова-

тельно «прокручивать» месяцы. Под ним располагается строка с названиями дней недели, после чего следует собственно блок чисел. В самом низу находится строка, отображающая текущую дату, которая, к тому же будет обведена в календаре. Впрочем, как строку с текущей датой, так и выделение дня можно отключить. В то же время, можно наоборот, добавить еще вывод номеров недель (как в квартальном календаре), или ограничить диапазон допустимых значений. Все это возможно благодаря специальным свойствам этого компонента, унаследованным от класса TCommonCalendar и перечисленным в таблице 13.2.

Таблица 13.2. Свойства компонента MonthCalendar

Свойство	Тип	Описание
CalColors	TMonthCalColors	Позволяет определить цвета для различных элементов календаря
Date	TDate	Определяет дату, являющуюся выбранной в календаре, или первую дату в диапазоне, если выбран диапазон
EndDate	TDate	Указывает на последнюю дату, являющуюся выбранной, если выбран диапазон дат
FirstDayOfWeek	TCalDayOfWeek	Определяет первый день недели. Значение по умолчанию берется из региональных настроек Windows (для России - понедельник)
MaxDate	TDate	Определяет максимальную дату, которая будет доступна в календаре
MaxSelectRange	Integer	Определяет максимальное количество дней, которые можно выбрать в качестве диапазона дат
MinDate	TDate	Определяет минимальную дату, которая будет доступна в календаре
MultiSelect	Boolean	Определяет возможность выбора диапазона дат
ShowToday	Boolean	Определяет, должна ли быть видна строка, показывающая текущую дату
ShowTodayCircle	Boolean	Определяет, должен ли текущий день быть отмечен в календаре
WeekNumbers	Boolean	Определяет, должны или нет отображаться номера недель

Если отбросить все оформительские свойства, то наиболее пристального внимания заслуживают, прежде всего, свойство Date, а так же EndDate, если разрешен выбор диапазона дат (что достигается путем присваивания истины свойству MultiSelect). Для диапазонов важно так же свойство MaxSelectRange – по умолчанию оно установлено в 31, но, установив его, скажем, в 7, можно ограничить выбор одной неделей.

Другой компонент, предназначенный для отображения и ввода дат – это DateTimePicker. И хотя внешне этот компонент больше похож на ниспадающий список, на самом деле он также является потомком класса TCommonCalendar, хотя и не позволяет обращаться ко всем его свойствам. Секрет заключается в том, что если поместить такой компонент на форму, то он действительно выглядит как раскрывающийся список, с тем лишь исключением, что вместо текста будет отображать дату. Но если запустить приложение и щелкнуть по кнопке раскрытия, то вместо списка мы увидим как раз тот самый календарь (рис. 13.3).

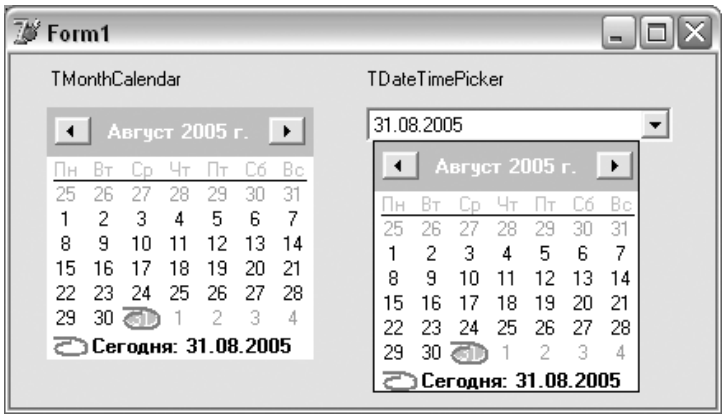

Рис. 13.3. Компонент MonthCalendar и компонент DateTimePicker в раскрытом состоянии

Вместе с тем, для компонента DateTimePicker доступны только такие свойства, как CalColors, Date, DateTime, MaxDate и MinDate. Это вызвано, прежде всего, тем, что он не поддерживает возможности работы с диапазонами дат. В то же время, он имеет и ряд иных свойств, и помимо всего прочего, может быть использован для ввода не только дат, но и времени (табл. 13.3).

Таблица 13.3. Собственные свойства DateTimePicker

Свойство	Тип	Описание
CalAlignment	TDTCalAlignment	Определяет выравнивание раскрывающегося календаря - по левому (dtaLeft) или по правому (dtaRight) краю
Checked	Boolean	Указывает, отмечен ли переключатель (наличие переключателя определяется свойством ShowCheckbox)
DateFormat	TDTDateFormat	Определяет формат вывода даты - короткий (dfShort) или полный (dfLong)
DateMode	TDTDateMode	Определяет способ ввода даты - при помощи ниспадающего календаря (dmUpDown) или только стрелками (dmComboBox)
DroppedDown	Boolean	Указывает, раскрыт ли календарь в данный момент
Format	String	Позволяет указать произвольный формат отображения дат
Kind	TDateTimeKind	Определяет тип вводимых данных - дата (dtkDate) или время (dtkTime)
ShowCheckbox	Boolean	Определяет, должен ли отображаться переключатель рядом с датой

Следует отметить, что свойство DateFormat имеет смысл только тогда, когда свойство Kind установлено в формат Date. А если элемент используется для ввода времени, то DateMode всегда будет иметь значение dmComboBox. При этом, когда установлен режим dmComboBox, все свойства, связанные с календарем так же теряют смысл.

Отдельно следует выделить свойство Format, отвечающее за вид выводимых значений. Например, используя собственный формат можно одновременно отобразить и

дату, и время. Для того, чтобы задать формат вывода, следует определить шаблон, руководствуясь данными, приведенными в таблице 13.4.

Таблица 13.4. Шаблоны формата вывода даты и времени для свойства Format

Маска	Описание
d	Число 1 или 2 цифрами
dd	Число 2 цифрами. Однозначные значения дополняются 0
ddd	Название дня недели в сокращенной форме
dddd	Название дня недели полностью
h	Час 1 или 2 цифрами, в 12-часовом формате
hh	Час 2 цифрами, в 12-часовом формате. Однозначные значения дополняются 0
H	Час 1 или 2 цифрами, в 24-часовом формате
HH	Час 2 цифрами, в 24-часовом формате. Однозначные значения дополняются 0
m	Минута 1 или 2 цифрами
mm	Минута 2 цифрами. Однозначные значения дополняются 0
M	Месяц 1 или 2 цифрами
MM	Месяц 2 цифрами. Однозначные значения дополняются 0
MMM	Название месяца в сокращенной форме
MMMM	Название месяца полностью
t	Часть дня (AM/PM) 1 буквой - A или P
tt	Часть дня (AM/PM) 2 буквами - AM или PM
yy	Последние 2 цифры года
yyyy	Год полностью

Кроме того, можно даже добавить какой-либо поясняющий текст. Например, если в качестве шаблона указать «Сегодня dddd, dd MMMM yyyy г.», то дата будет отображаться как «Сегодня четверг, 1 сентября 2005 г.».

В дополнение к дате можно добавить время, например, благодаря шаблону типа «dd/MM/yy HH:mm» будет выводиться информация вида «01/09/05 12:30». При этом следует учитывать, что к дате следует обращаться через свойство Date, а ко времени – через свойство Time. Программно также доступно свойство DateTime, при помощи которого можно установить сразу и дату, и время:

```
DateTimePicker1.DateTime:=now();
```

Здесь функция now возвращает текущую дату и время по часам компьютера, и присваивает это значение свойству DateTime компонента DateTimePicker. При этом автоматически обновятся оба его свойства, отвечающие за хранение даты и времени отдельно, т.е. Date и Time.

Строка состояния

Многие приложения имеют строку состояния – полосу внизу главного окна, отображающую состояние наиболее важных для данной программы индикаторов. Для текстового редактора это будут режимы ввода с клавиатуры и позиция каретки, для браузера – состояние загрузки страницы, и т.д. Для этих целей можно использовать обычную панель и размещенные на ней текстовые подписи или иные элементы инди-

кации, однако в большинстве случаев гораздо удобнее воспользоваться специализированным компонентом – StatusBar.

Компонент StatusBar может быть использован в 2 основных режимах – в простом, когда вся его поверхность используется для вывода одного сообщения, и в многопанельном, кодга он разделяется на отдельные области, предназначенные для вывода различной информации – панели. За режим работы отвечает свойство SimplePanel, установив которое в значение истины, мы получим простой режим. При этом текст надписи, выводимой в строку состояния, определяется свойством SimpleText.

В случае, когда необходимо вывести несколько индикаторов, для деления строки состояния на отдельные панели следует использовать свойство Panels, представляющее собой массив элементов типа TStatusPanel. Основными свойствами такой панельки являются Width и Text. Первое задает ее ширину, а второе – содержимое надписи. Так же могут пригодиться свойства Alignment и Bevel. Свойство Alignment определяет выравнивание текста в панельке, а свойство Bevel задает вид рамки. Свойство Bevel может принимать значения pbLowered, pbRaised и pbNone, задавая вогнутую, выпуклую или вообще отсутствующую рамку, соответственно.

Для создания новых панелек можно использовать инспектор объекта. Для этого достаточно щелкнуть по кнопке в строке со свойством Panels и в открывшемся окне редактирования панелек нажать кнопку Add New. При этом в списке редактирования появится новый элемент, щелкнув по которому, вы получите доступ ко всем его свойствам в инспекторе объекта.

Используя инспектор объекта, удобно создавать панельки, а при необходимости можно также задать начальные значения надписей. Но поскольку панель состояния по своей природе не должна быть статичной, то в процессе выполнения программы придется обращаться к отдельным ее элементам программно. Делается это через массив Panels, каждый элемент которого, как мы уже знаем, является объектом типа TStatusPanel:

```
StatusBar1.Panels[0].Text:='Программа загружена!'; //текст на 1-ой панельке
StatusBar1.Panels[0].Width:=150; // изменяем ширину
StatusBar1.Panels[0].Alignment:=taCenter; // выравниваем текст по центру
StatusBar1.Panels[1].Text:=''; // удаляем текст 2-й панельки
```

Как и у других оконных элементов, оформление шрифта для строки состояния определяется при помощи свойства Font. Однако его изменения не будут иметь силы до тех пор, пока свойство UseSystemFont имеет значение истины. Дело в том, что для строки состояния используется шрифт, определенный в качестве системного в настройках Windows. При помощи свойства Font вы можете его переопределить для своей строки состояния, но чтобы задействовать изменение, требуется задействовать свойство UseSystemFont и установить его в ложь.

Наконец, строка состояния обладает еще одним, весьма полезным свойством – AutoHint. Если установить его значение в истину, то в качестве текста строки состояния (или текста ее первой панели для многопанельного варианта) будет отображаться подсказка (свойство Hint) к тому элементу, над которым находится указатель мышки. Это может быть использовано в качестве альтернативы, если показ подсказок (свойство ShowHint) у всех элементов окна отключено, или как дополнение к всплывающим подсказкам.

Панель инструментов

Строка состояния – далеко не единственный часто используемый элемент интерфейса в современных приложениях. Не меньшее, а, возможно, даже большее их число обладает другим обязательным элементом интерфейса – панелью инструментов. Опять-таки, для этих целей можно использовать обычную панель с размещенными на ней обычными же кнопками. Но для создания инструментальной панели (tool bar) удобнее использовать компонент, который так и называется – ToolBar.

Инструментальная панель, в первую очередь, предназначена для размещения специальных кнопок инструментальной панели – ToolButton. Вместе с тем, на нее можно поместить практически любой другой компонент, включая обычные кнопки, списки, текстовые редакторы и т.д. Свойства этого компонента приведены в таблице 13.5.

Таблица 13.5. Свойства компонента ToolBar

Свойство	Тип	Описание
HideClippedButtons	Boolean	Определяет обработку кнопок, затененных смежной панелью инструментов
ButtonCount	Integer	Указывает количество кнопок, расположенных на панели
ButtonHeight	Integer	Определяет высоту элементов управления, расположенных на панели
Buttons	array of TToolButton	Содержит массив всех кнопок, имеющихся на панели
ButtonWidth	Integer	Определяет высоту кнопок, расположенных на панели
Customizable	Boolean	Определяет возможность настройки панели пользователем
CustomizeKeyName	string	Определяет имя раздела в реестре, в котором будет сохраняться информация о пользовательских настройках
CustomizeValueName	string	Определяет имя записи в реестре, в которой будет сохраняться информация о пользовательских настройках
DisabledImages	TCustomImageList	Определяет коллекцию картинок для неактивного состояния кнопок (т.е. когда кнопка недоступна)
Flat	Boolean	Определяет, долена ли панель быть прозрачной (в стиле MSIE 3)
HotImages	TCustomImageList	Определяет коллекцию картинок для активного состояния кнопок (т.е. в момент, когда указатель мышки находится над кнопкой)
Images	TCustomImageList	Определяет коллекцию картинок для обычного состояния кнопок
Indent	Integer	Определяет поле по левой стороне панели
List	Boolean	Определяет вид выравнивания картинок и текста
Menu	TMainMenu	Позволяет связать панель инструментов с главным меню
RowCount	Integer	Указывает на количество строк, по которым рас-

		положены кнопки панели
ShowCaptions	Boolean	Определяет, должны или нет отображаться текстовые подписи к кнопкам (т.е. их Caption)
Transparent	Boolean	Определяет, будет ли панель прозрачной. Не оказывает влияния на сами кнопки
Wrapable	Boolean	Определяет, должны ли кнопки автоматически переноситься, если не помещаются в одну строку
EdgeBorders	TEdgeBorders	Определяет, по каким сторонам панели должны быть видны рамки
EdgeInner	TEdgeStyle	Определяет вид внутренних рамок. Допустимые значения: esNone, esRaised и esLowered
EdgeOuter	TEdgeStyle	Определяет вид внешних рамок. Допустимые значения: esNone, esRaised и esLowered

Наиболее важным для панели инструментов свойствами является такое свойство, как Images, поскольку именно коллекция картинок определяет вид кнопок. Размеров кнопок, находящихся на панели инструментов, всегда одинаковые, и задаются при помощи свойств ButtonHeight и ButtonWidth. При этом следует учитывать, что высота кнопок должна быть меньше, чем высота самой панели. Немаловажное влияние оказывают на вид панели свойства List и ShowCaption. Так, по умолчанию показ надписей отключен. Если же его включить, то текст надписи будет располагаться под картинкой, и все это будет выровнено по центру. Если же задействовать свойство List, установив его в истину, то текст и картинка будут расположены в строку, при этом текст будет справа, а картинка – слева.

Что касается собственно кнопок, типа ToolButton, то вы не найдете такого компонента на палитре инструментов. Зато их можно добавить, воспользовавшись контекстным меню панели, для чего следует выбрать пункт New Button. Самые важные свойства кнопок – это ImageIndex, определяющее номер картинки в коллекции изображений, ассоциированной с панелью, на которой находится данная кнопка, и Caption, определяющее ее текстовую подпись. Впрочем, эта самая подпись не будет видна, если только для самой панели не установлено в истину свойство List.

Кроме кнопок, на панелях инструментов часто используют разделители. Разделителями могут выступать те же кнопки, только специальным образом оформленные. Для помещения разделителя на панель из ее контекстного меню следует выбрать пункт New Separator.

Основным отличием между «просто кнопкой» и разделителем является то, что если у стандартной кнопки свойство Style установлено в значение tbsButton, то у разделителя – в tbsSeparator. Такая кнопка-разделитель в работающей программе выглядит кК пустое пространство между соседними кнопками. Если же требуется отделить группы кнопок более явно, то можно использовать другой стиль кнопки-разделителя, установив свойство Style в tbsDivider, что добавит разделителю вертикальную черту.

Помимо 3 уже рассмотренных значений, для свойства Style предусмотрено еще 2: tbsCheck и tbsDropDown. Оба они позволяет изменять вид кнопки, причем если выбран вариант tbsCheck, то она получает триггерный эффект, т.е. «залипает» при первом нажатии, и «отлипает» при следующем. Но если таких кнопок несколько, и для них еще одно свойство – Grouped – установлено в истину, то выбранной в один мо-

мент времени всегда будет только одна из кнопок группы, а все остальные будут автоматически возвращаться в обычное состояние. Текущее состояние кнопки можно узнать, обратившись к ее свойству Down: если оно установлено в истину, то кнопка нажата («включена»), если в ложь – отжата.

Что касается стиля tbsDropDown, то он используется для создания комбинированной кнопки-меню. При этом должно быть определено другое свойство – DropdownMenu, которое ассоциирует кнопку с каким-либо имеющимся на форме контекстным меню (компонент PopupMenu).

Чтобы лучше разобраться с кнопками, создадим небольшое демонстрационное приложение. Для этого создадим новый проект и поместим на форму компонент ToolBar. Из группы Win32 нам понадобится еще один компонент – ImageList, а из группы стандартных (на закладке Standard) – PopupMenu. Теперь создадим кнопки. Для этого, используя контекстное меню компонента ToolBar (т.е. щелкая по нему правой клавишей мышки), последовательно добавим 9 элементов в такой последовательности: 1-й – кнопка (Add Button), 2-й – разделитель (Add Separator), 3-й – кнопка, 4-й – разделитель, 5-й – кнопка, 6-й – разделитель, и оставшиеся 3 – кнопки. Теперь для 2-го и 3-го разделителей, при помощи инспектора объекта, установим стиль tbsDivider. Затем для кнопки, оказавшейся между разделителями с вертикальной чертой, установим стиль tbsDropDown. После этого выберем все 3 последние кнопки, для чего используем клавишу Shift, и установим для них свойство Style в tbsCheck, а Grouped – в истину. В результате панель инструментов примет вид, показанный на рис. 13.4.

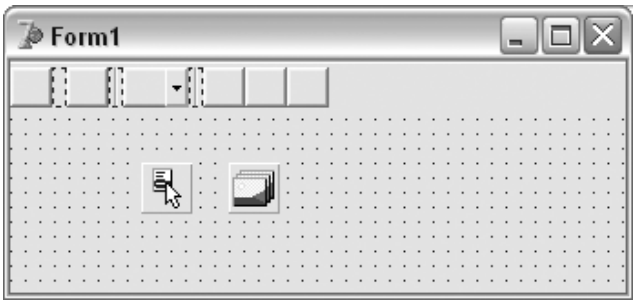

Рис. 13.4. Панель инструментов с кнопками различных видов

Теперь следует заняться свойствами самой панели, для чего, в первую очередь, следует определить изображения. Для этого надо будет воспользоваться редактором компонента ImageList, добавив в него 6 изображений – по числу имеющихся у нас кнопок. После этого свойству Images панели инструментов присвоим значение ImageList1, т.е. коллекцию картинок. Сразу после этого все кнопки получат соответствующие иконки.

Чтобы визуально отделить панель от остальной части окна, в ее составном свойстве EdgeBorders следует установить для ebBottom значение True. Таким образом, само свойство EdgeBorders получит значение [ebTop,ebBottom], а рамка будет видна не только по верхней, но и по нижней границе панели.

Остается разобраться только с той кнопкой, для которой мы установили стиль tbsDropDown. Прежде всего, надо определить меню, которое будет раскрываться при щелчке пользователем по стрелке. Для этого мы уже поместили на форму компонент

PopupMenu, так что надо лишь создать в нем несколько пунктов. После этого необходимо ассоциировать это меню с кнопкой, для чего в инспекторе объекта найдем для данной кнопки свойство DropDownMenu и установим в качестве его значения имя контекстного меню – PopupMenu1.

Результатом проделанной работы будет приложение, имеющее одну раскрывающуюся и 3 триггерные кнопки, которое можно найти в каталоге Demo\Part3\Toolbar.

Попутно следует отметить, что в том случае, если требуется использовать несколько панелей инструментов, и при этом дать пользователю возможность еще и перемещать их, то в качестве основы берется компонент ControlBar (расположен на закладке Additional палитры компонентов), и все панели помещаются не непосредственно на форму, а не этот компонент.

ПРИМЕЧАНИЕ

> Помимо компонента ToolBar, в Delphi имеется еще один компонент, предназначенный для создания панелей инструментов – CoolBar. Он совмещает в себе контейнер для панелей и сами панели.

Элементы с вкладками

Элементы с вкладками представляют собой элементы управления, состоящие из нескольких страниц (листов, вкладок), на которые можно переходить, щелкая мышкой по их ярлыкам. Всего имеется 2 таких элемента – это компоненты TabControl и PageControl, причем оба они происходят от класса TCustomTabControl, и, соответственно, имеют ряд общих свойств и методов. И тот, и другой компоненты являются контейнерами для других элементов, хотя организация и управление этими элементами имеют ряд различий. Основное различие заключается в том, что TabControl представляет собой одну общую панель, имеющую закладки, в то время, как в PageControl каждая закладка является частью отдельной панели.

Но начнем все-таки с общих черт этих компонентов, для чего обратимся к таблице 13.6, содержащей все основные свойства их предка – класса TCustomTabControl.

Таблица 13.6. Общие свойства TabControl и PageControl

Свойство	Тип	Описание
HotTrack	Boolean	Определяет, должны ли надписи закладок выделяться цветом при наведении указателя мышки
Images	TCustomImageList	Указывает на коллекцию изображений, ассоциированную с данным элементом и используемую для оформления закладок
MultiLine	Boolean	Определяет, могут ли закладки размещаться в несколько рядов, если не помещаются в 1 ряд
RaggedRight	Boolean	Определяет, должны ли закладки растягиваться таким образом, чтобы заполнить все пространство по ширине элемента
ScrollOpposite	Boolean	Определяет вариант перемещения линий с закладками при выборе закладки в случае многорядного размещения
Style	TTabStyle	Определяет вид закладок. Допустимые значе-

		ния: tsTabs (обычные), tsButtons (в виде кнопок) и tsFlatButtons (в виде плоских кнопок)
TabHeight	Smallint	Определяет высоту закладок в пикселях
TabIndex	Integer	Указывает на порядковый номер выбранной в настоящий момент закладки
TabPosition	TTabPosition	Определяет расположение закладок. Допустимые значения: tpTop, tpBottom, tpLeft и tpRight
TabWidth	Smallint	Определяет фиксированную ширину закладок в пикселях. Если установлено в 0, то ширина будет назначаться автоматически

Вместе с тем, класс TCustomTabControl, имеет еще 2 свойства, которые присутствуют у компонента TabControl, но которых нет у PageControl. Это – свойство Tabs, содержащее список названий всех закладок, и свойство MultiSelect, отвечающее за возможность выделения нескольких закладок одновременно. То, что данные свойства не поддерживаются для PageControl, объясняется их принципиальными отличиями. Зато у него имеется собственное свойство Pages, являющееся хранилищем самих страниц с закладками, а отсутствие свойства MultiSelect объясняется тем, что невозможно выбрать (и отобразить) несколько страниц одновременно.

Кроме свойства Pages, у компонента PageControl имеются еще несколько индивидуальных свойств. В частности, свойство PageCount указывает на количество страниц, а ActivePageIndex – на порядковый номер выбранной страницы. Тем самым оно, в типичном случае, дублирует свойство TabIndex, поскольку закладка является частью страницы, за тем лишь исключением, если закладки у части страниц не отображаются. Еще одно свойство – ActivePage ссылается на саму страницу как на компонент типа TabSheet.

Сам компонент TabSheet используется только как составная часть элемента PageControl. Его собственные свойства позволяют идентифицировать страницу среди других закладок. Так, свойство PageIndex позволяет узнать, какой по счету является данная страница, или же изменить ее расположение, а свойство PageControl ссылается на родительский элемент. Свойство ImageIndex позволяет назначить представляющей страницу закладке изображение, ссылаясь на его номер в коллекции изображений, ассоциированной с родительским компонентом. А при помощи TabVisible можно определить, должна или нет быть видна закладка.

Что касается работы с отдельными страницами, то для того, чтобы добавить новую страницу к компоненту PageControl, достаточно воспользоваться его контекстным меню и выбрать пункт New Page. При помощи этого же контекстного меню можно и удалить ненужные страницы. В то же время, закладки для компонента TabControl определяются другим способом – путем указания их перечня в свойстве Tabs, для чего следует использовать инспектор объекта. Если же требуется выполнить какие-либо манипуляции со страницами во время выполнения программы, то, опять-таки для TabControl это можно делать путем непосредственной правки списка Tabs:

```
TabControl1.Tabs.Add('Новая');
TabControl1.Tabs.Delete(1);
```

В то же время, для PageControl не все так просто, поскольку каждая его закладка является отдельным объектом вкупе со страницей, частью которой она является. Соответственно, чтобы создать новую страницу, для начала необходимо создать соответ-

ствующий объект, после чего ассоциировать его с нужным элементом PageControl при помощи одноименного свойства:

```
NewTabSheet: TTabSheet;
...
NewTabSheet := TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
```

В отличие от свойств, среди методов 2 рассматриваемых компонентов, общих наберется весьма немного. Среди них можно выделить лишь RowCount, возвращающий число строк, по которым размещены закладки, и ScrollTabs, позволяющий «прокрутить» закладки в случае, если они все находятся в 1 ряду, и полностью на нем не помещаются. Основное же число методов приходится на компонент PageControl, однако многие из них могут быть интересны лишь для детального изучения работы самого компонента.

Из реально востребованных методов можно отметить лишь связанные с программной навигацией по закладкам. В частности, метод FindNextPage позволяет получить доступ к странице, предшествующей, или следующей за текущей. Они имеет следующий синтаксис:

```
function FindNextPage(CurPage: TTabSheet; GoForward, CheckTabVisible:
Boolean): TTabSheet;
```

Первый параметр указывает на текущую страницу, второй определяет направление перемещения (истина – вперед, ложь – назад), а последний определяет, должны ли игнорироваться страницы, свойство TabVisible которых установлено в ложь (т.е. закладки которых не отображаются).

Второй метод – SelectNextPage – позволяет изменить выбранную страницу, отталкиваясь от той, которая выбрана в настоящий момент. В целом он сходен с только что рассмотренным методом FindNextPage, но, поскольку отправная точка уже известна, а результатом его работы является лишь переключение страниц, то ни текущей страницы, ни возвращаемого значения у него нет. В результате, мы имеем процедуру следующего вида:

```
procedure SelectNextPage(GoForward: Boolean; CheckTabVisible: Boolean=True);
```

То, что последний параметр является предопределенным, говорит о том, что его можно опускать. Но если его все-таки указать и установить в значение лжи, то в том случае, если закладки для некоторых страниц действительно не отображаются, это будет единственным способом обеспечить навигацию по «скрытым» страницам.

Прочие компоненты Win32 и стиль Windows XP

Пожалуй, наиболее сложными визуальными компонентами среди всех элементов 32-разрядного интерфейса, являются TreeView и ListView. Чтобы наглядно представить себе, что они из себя представляют, запустите Проводник Windows. Фактически, левая его часть, отображающая все места хранения данных на ПК в виде иерархической структуры, иллюстрирует TreeView, а правая, с файлами и папками – ListView. При этом, подобно виду в Проводнике, компонент ListView в Delphi может отображать свое содержимое в виде списка, иконок, или таблицы. Каждый элемент представляет собой объект, имеющий собственную подпись, значок, воженные и ассоциированные объекты.

Подробнее ознакомиться с этими компонентами можно на примере приложения Resource Explorer, входящем в поставку Delphi (он находится в подкаталоге Demos\ResXplor, вложенном в тот каталог, в который производилась установка самой Delphi, например, C:\Program Files\Delphi 7\).

В противоположность компонентам TreeView и ListView, компонент XPManifest, появившийся в Delphi 7, можно назвать самым простым компонентом, причем не только среди элементов 32-разрядного интерфейса, но и вообще в VCL. Для него не имеется ни событий, ни методов, а определены лишь 2 свойства, обязательных для любого компонента – Tag и Name. Фактически, его предназначение сводится к тому, что при помещении на форму, к проекту подключается модуль XPMan, который, в свою очередь, подключает файл ресурсов WindowsXP.res. Результатом помещения компонента XPManifest на форме приложения будет то, что интерфейс программы при работе в Windows XP будет «понимать» темы оформления Windows XP, и все его компоненты во время выполнения программы будут выглядеть так, как это принято в Windows XP (рис. 13.5).

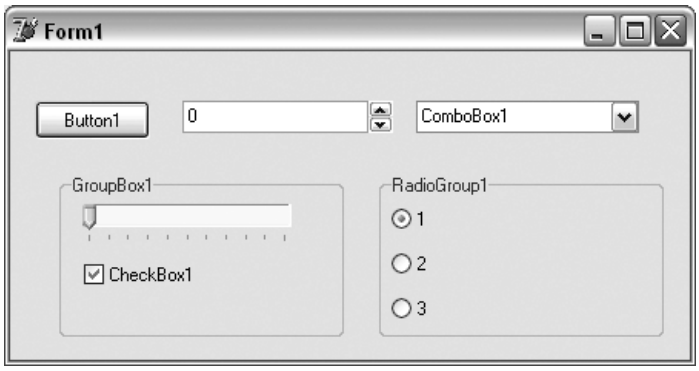

Рис. 13.5. Интерфейс приложения, скомпилированного в Delphi 7 с использованием XPMan

Если же вы работаете с более ранней версией Delphi, или же не будете включать в состав приложения модуль XPMan, то вид элементов управления будет таким, как в предыдущих версиях Windows (рис. 13.6).

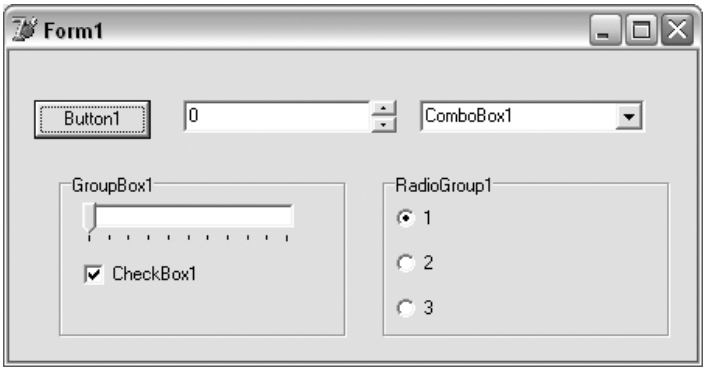

Рис. 13.6. Интерфейс того же самого приложения, без использования модуля XPMan

В то же время, существует возможность придать виду приложения современный вид и при использовании версий Delphi, появившихся до выхода Windows XP. Для этого достаточно создать специальный файл формата XML с расширением manifest и названием, совпадающим с полным именем выполняемого файла приложения. Например, если исполняемый файл называется D6.exe, то файл, декларирующий возможность использования стилей Windows XP, будет называться D6.exe.manifest. Что касается его содержимого, то оно должно быть таким, как показано в листинге 13.1.

Листинг 13.1. XML-файл для использования стилей Windows XP

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity version="5.1.0.0" name="Microsoft.Windows.Shell.Gina"
processorArchitecture="*" type="win32"/>
 <dependency>
  <dependentAssembly>
   <assemblyIdentity type="win32" name="Microsoft.Windows.Common-Controls"
version="6.0.0.0" language="*" processorArchitecture="*"
publicKeyToken="6595b64144ccf1df" />
  </dependentAssembly>
 </dependency>
</assembly>
```

В то же время, если применить такой файл к приложению, скомпилированному при помощи Delphi 7 без использования компонента XPManifest, то его вид ничем не будет отличаться от такового, созданного с использованием этого компонента. Несколько по-другому дела обстоят с предыдущими версиями Delphi, поскольку библиотека VCL ранних версий, разумеется, не обеспечивала поддержку теперешних стилей. В результате 2рестайлингу» будут подвержены только те компоненты, которые являются прямой проекцией стандартных компонентов Windows, т.е. только простые кнопки, текстовые поля и компоненты 32-разрядного интерфейса (рис. 13.7).

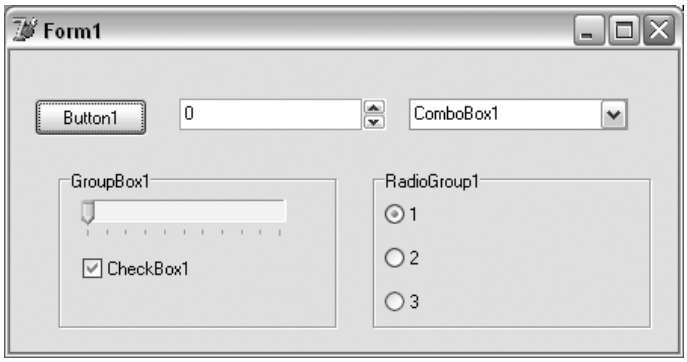

Рис. 13.7. Приложение Delphi 6 с файлом manifest

Что касается компонентов, не имеющих прямых аналогов в самой Windows (различные группирующие элементы из группы Standard, а так же любые элементы, не относящиеся к компонентам Standard и Win32), будут отображаться в классическом стиле.

Примеры приложения, скомпилированного в разных версиях Delphi, с использованием XPManifest и без, можно найти в каталоге Demo\Part3\XPman.

Стандартные диалоги и редактор RTF

Прежде, чем продолжить свое знакомство с визуальными компонентами, рассмотрим невизуальные, в первую очередь те, что предназначены для обращения к стандартным сервисным диалогам Windows. В частности, это диалоги открытия и сохранения файлов, диалоги выбора цвета и шрифта, а так же диалоги печати, поиска и замены. Все они должны быть вам хорошо знакомы, поскольку используются практически повсеместно в Windows-приложениях. Но чтобы лучше разобраться с их внутренним устройством, мы так же рассмотрим один визуальный компонент, для которого все рассматриваемые диалоги являются актуальными, а именно – редактор форматированного текста, для чего мы даже создадим свой текстовый редактор.

Диалоги работы с файлами

Все диалоги собраны на обзей закладке Dialogs палитры компонентов. Первым на ней является компонент OpenDialog, представляющий собой оболочку для функций Windows API, связанных с диалогом выбора файлов. Диалог, вызываемый при помощи компонента OpenDialog, используется для выбора (открытия) файла. Для операции сохранения следует использовать другой компонент – SaveDialog. Он показывает, в общем-то, тот же самый диалог, но в режиме сохранения.

Еще 2 компонента – OpenPictureDialog и SavePictureDialog являются частным случаем стандартных диалогов открыть-сохранить, но предназначенными для работы с графическими файлами. Визуально они отличаются от универсальных диалогов тем, что имеют область просмотра изображения. С точки зрения их использования в программе они ничем не отличаются от OpenDialog и SaveDialog.

Вообще, следует отметить, что базовым классом для всех диалогов выступает TCommonDialog, а для всех диалогов работы с файлами – класс TOpenDialog. Итак, рассмотрим общие свойства файловых диалогов на основе OpenDialog, для чего обратимся к таблице 14.1.

Таблица 14.1. Свойства файловых диалогов

Свойство	Тип	Описание
DefaultExt	String	Определяет расширение по умолчанию, которое будет добавляться к файлу, если пользователь не указал расширение явно
FileName	String	Определяет имя файла, включая его путь
Files	TStrings	Содержит список выбранных файлов, если таковых несколько (задается свойством Options путем включения флага ofAllowMultiSelect)
Filter	String	Определяет строку, содержащую фильтр типов файлов
FilterIndex	Integer	Определяет, какой из вариантов фильтра должен быть выбран по умолчанию
InitialDir	String	Определяет каталог, который будет отображен изначально
Options	TOpenOptions	Определяет вид особенности поведения диалога
OptionsEx	TOpenOptionsEx	Определяет дополнительные опции, актуальные для Windows Me, 2000 и более новых версий
Title	String	Определяет текст, который будет отображен в заголовке диалога

Наиболее востребованным свойством любого файлового диалога, разумеется, являются свойство FileName, поскольку именно оно определяет имя файла, выбранного пользователем. В то же время для настроек диалога важны и другие свойства, в частности, при помощи свойства Filter задают маску, по которой пользователь сможет фильтровать типы файлов. И хотя это свойство представляет собой строку, для его правки при посредстве инспектора объектов используется табличный редактор. Более того, для специализированных диалогов, предназначенных для работы с графическими файлами, это свойство заполняется автоматически (рис. 14.1).

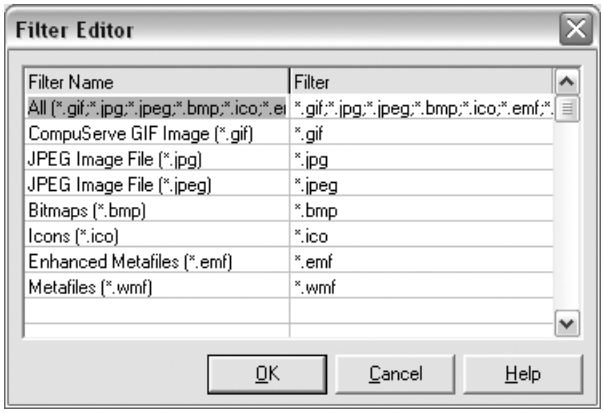

Рис. 14.1. Редактор свойства Filter с данными по умолчанию для OpenPictureDialog

В столбце слева указывают название фильтра (т.е. то, что видит пользователь), а в столбце справа – собственно шаблон фильтра. Если же требуется задать фильтр программно, то строку разделяют при помощи символа вертикальной черты. В таком случае сначала указывают название, потом ставят черту, а затем – определяют шаблон. Для следующего варианта все повторяют, при этом сами варианты так же отделяются вертикальной чертой:

```
OpenDialog1.Filter:='Все файлы|*.*|Текстовые файлы|*.txt';
```

Здесь мы определили 2 варианта фильтра – для отображения всех файлов (шаблон *.*) и для отображения только текстовых файлов (шаблон *.txt). Если один шаблон включает в себя несколько разных расширений, то они перечисляются через точку с запятой:

```
OpenDialog1.Filter:='Все файлы|*.*|Файлы Delphi|*.pas;*.dpr';
```

Еще одно важное свойство файловых диалогов – это Options. Именно при помощи множества флагов этого свойства определяют особенности поведения диалога, а также устанавливают отображение дополнительных элементов его интерфейса. Значения всех актуальных флагов приведены в таблице 14.2.

Таблица 14.2. Значения флагов свойства Options

Флаг	**Описание**
ofReadOnly	Делает опцию "только чтение" включенной по умолчанию
ofOverwritePrompt	Указывает на необходимость вывода предупреждающего сообщения, если файл с указанным именем уже существует (для диалогов сохранения)

ofHideReadOnly	Удаляет из диалога переключатель "только чтение"
ofNoChangeDir	Возвращает исходный путь после закрытия диалога
ofShowHelp	Отображает кнопку справки на диалоге
ofNoValidate	Отключает проверку на недопустимые символы в имени файла
ofAllowMultiSelect	Позволяет пользователю выбрать несколько файлов одновременно
ofExtensionDifferent	Этот флаг включается автоматически, когда выбранный файл имеет расширение, отличное от указанного в свойстве DefaultExt
ofPathMustExist	Выдает сообщение об ошибке, если указанного пользователем пути не существует
ofFileMustExist	Выдает сообщение об ошибке, если указанного пользователем файла не существует (для диалогов открытия файла)
ofCreatePrompt	Выдает сообщение с предупреждением о необходимости создания файла, если указанного пользователем файла не существует
ofShareAware	Позволяет игнорировать ошибки доступа
ofNoReadOnlyReturn	Выдает сообщение об ошибке, если пользователь выберет файл, доступный только для чтения
ofNoTestFileCreate	Отключает проверку на права доступа к сетевым ресурсам
ofNoDereferenceLinks	Отключает обработку ярлыков. Т.е. если этот флаг не установлен, то выбор ярлыка приведет к выбору файла, на который ярлык ссылается, в противном случае будет выбран сам файл ярлыка (lnk)
ofEnableSizing	Позволяет пользователю изменять размеры диалогового окна (не действует в Windows 95 и Windows NT 4.0)
ofDontAddToRecent	Не позволяет заносить выбранные файлы в список недавно открытых документов
ofForceShowHidden	Позволяет пользователю увидеть скрытые файлы

По умолчанию включены только ofHideReadOnly и ofEnableSizing, т.е. скрыта редко используемая на практике опция открытия файлов в режиме «только чтение», и разрешено изменение размеров диалогового окна.

Что касается второго свойства – OptionsEx, то для него пока что доступна единственная настройка – ofExNoPlacesBar, позволяющая отключить боковую панель быстрого доступа к основным разделам компьютера, поддерживаемую в Windows Me, 2000, XP и 2003 Server.

Еще проще дела обстоят с методами. Кроме стандартных методов, доставшихся диалогам от класса TComponent, у него всего один собственный метод – Execute. Именно при помощи этого метода вызываются диалоги в коде программы. Метод Execute возвращает булевское значение – истину, если пользователь нажмет OK, или ложь, если пользователь выберет отмену. Соответственно, типичный вызов диалога в программе выглядит следующим образом:

```
if OpenDialog1.Execute then begin
  ...
end;
```

В том же случае, если единственным предназначением процедуры или функции, обращающейся к диалогу, является получение имени файла (например, если это обработчик для кнопки типа «открыть файл»), то удобнее использовать отрицание:

```
procedure TForm1.OpenFileButtonClick(Sender: TObject);
begin
  if not OpenDialog1.Execute then exit;
  Form1.Caption:=OpenDialog1.FileName;
  Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
end;
```

Такой подход позволяет избежать лишнего блока begin-end, поскольку если диалог не выполнен (т.е. пользователь передумал открывать файл), то можно сразу же выйти из подпрограммы, предназначенной для обработки загрузки файла.

Диалоги печати

Для взаимодействия со средствами Windows по подготовке к печати в VCL предусмотрено 3 компонента: PrintDialog, PrintSetupDialog и PageSetupDialog. Первый из них реализует доступ к стандартному диалогу печати, второй – к диалогу настройки печати, а третий – к диалогу параметров страницы.

Свойства, которыми обладают данные компоненты, в основном, предназначены для непосредственного обмена информацией между приложением и окном диалога, вернее, даже с его составными элементами. Так, для диалога печати (PrintDialog, рис. 14.2), это будут свойства, отвечающие за диапазон печати и число копий. В частности, за число копий отвечает свойство Copies, за тип диапазона, который может принимать одно из 3 значений – все (prAllPages), указанные страницы (prPageNums), или выделенный фрагмент (prSelection) – отвечает свойство PrintRange. При этом, если выбран диапазон по страницам, то его границы будут определяться свойствами FromPage и ToPage. Ограничить границы выбора этих значений пользователем во время его работы с диалогом можно при помощи свойств MaxPage и MinPage.

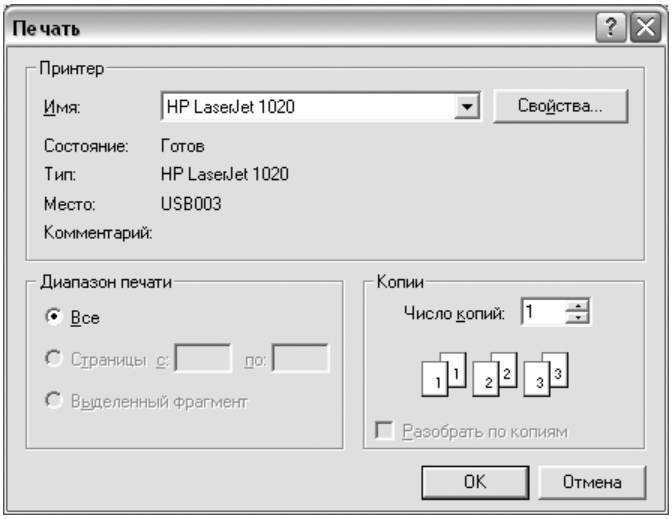

Рис. 14.2. Диалог печати

Как и у большинства других диалогов, диалог печати имеет свойство Options, при помощи которого можно настроить вид и поведение диалога. Приведем описание значений его флагов:

- poDisablePrintToFile – Отключает (делает недоступной) опцию печати в файл, актуально только если включен флаг poPrintToFile;
- poHelp – Отображает кнопку справки на окне диалога;
- poPageNums – Делает доступной опцию печати диапазона страниц;
- poPrintToFile – Отображает переключатель вывода в файл;
- poSelection – Делает доступной опцию печати выделенного фрагмента;
- poWarning – Производит проверку на доступность выбранного принтера, и выдает предупреждение, если устройство не доступно.

Важно осознавать, что сам по себе диалог печати ничего не делает. Фактически он лишь предоставляет пользовательский интерфейс для выбора диапазона печати и указания числа копий. Дальнейшую обработку полученной информации должно выполнять само приложение, для чего используется объект Printer (см. главу 9). В частности, объект «принтер» имеет свойство Copies, обозначающее количество копий, так что с этой точки зрения все просто, надо лишь не забыть включить модуль Printers в блок Uses, дальше останется лишь присвоить соответствующее свойство:

```
Printer.Copies:=PrintDialog1.Copies;
```

Ряд других свойств объекта Printer настраиваются при помощи диалога настроек печати, PrinterSetupDialog. Однако, в отличие от диалога печати, изменения, произведенные пользователем в диалоге настроек, а именно – размер бумаги, устройство подачи и ориентация страницы – будут применены автоматически. Во многом это связано с тем, что данные настройки напрямую зависят от аппаратного обеспечения: ведь вы не можете установить размер бумаги, не поддерживаемый принтером, или назначить в качестве подающего лотка устройство, которого нет физически.

Еще один диалог, имеющий отношение к печати – это диалог настройки параметров страницы, PageSetupDialog, появившийся в Delphi 7. С его помощью пользователь может указать ориентацию страницы (портретную или ландшафтную), установить размер листа и источник подачи, а также задать поля. Иначе говоря, этот диалог предоставляет больше возможностей для выбора параметров страницы, чем PrinterSetupDialog. Впрочем, при помощи свойства Options можно отключить те или иные настройки, как-то выбор размера листа, его ориентации, полей, или устройства подачи:

- psoDefaultMinMargins – Учитывает минимальные значения, заданные в свойствах для полей страницы;
- psoDisableMargins – Отключает элементы интерфейса, относящиеся к установке полей;
- psoDisableOrientation – Отключает элементы интерфейса, относящиеся к выбору ориентации страницы;
- psoDisablePagePainting – Отключает вывод демонстрационного рисунка;
- psoDisablePaper – Отключает элементы интерфейса, относящиеся к размеру и подаче бумаги;
- psoDisablePrinter - Делает недоступной кнопку выбора принтера;

- psoMargins – Устанавливает начальные значения полей в значения, заданные при помощи свойств диалога. В противном случае использует значения по умолчанию;
- psoMinMargins – Устанавливает минимальные значения полей в значения, заданные при помощи свойств диалога. В противном случае использует значения по умолчанию;
- psoShowHelp – Делает доступной кнопку вызова справки;
- psoWarning – Выводит предупреждение, если в системе не выбран принтер по умолчанию.

Что касается тех параметров, которые не назначаются принтеру автоматически (в частности, поля), то как раз они и представлены свойствами компонента PageSetupDialog. Это свойства MarginLeft, MarginTop, MarginRight и MarginBottom, отвечающие за поля слева, сверху, справа и снизу, соответственно. При этом существует еще и возможность ограничить минимальный размер полей, что делается путем включения флага psoDefaultMinMargins, а сами значения задаются при помощи свойств MinMarginLeft, MinMarginTop, MinMarginRight и MinMarginBottom. Типом данных для всех этих свойств является целое, однако вопрос состоит в том, что за единицы измерения они представляют. За этот вопрос отвечает свойство Units, которое может принимать 3 значения:

- pmDefault – используются единицы измерения, принятые в системе по умолчанию;
- pmInches – значения интерпретируются как сотые доли дюйма;
- pmMillimeters – значения интерпретируются как сотые доли миллиметра.

Эти же единицы применяются не только к полям, но и еще к 2 свойствам – PageWidth и PageHeight, представляющими собой размеры страницы в ширину и в высоту. Так, если свойство Units установлено в pmMillimeters, а в качестве носителя пользователь выберет лист размером A4, эти свойства примут значения 21000 и 29700, т.е. 210 мм и 297 мм.

Подобно диалогам для работы с файлами, диалоги печати имеют в своем распоряжении единственный собственный метод – Execute, который следует использовать для вызова окна диалога по ходу выполнения программы.

Диалог шрифта

Диалоги открытия файлов и печати можно отнести к группе ввода-вывода данных. Пожалуй, лишь диалог параметров страницы относится к их форматированию. В то же время, существует еще диалоги, ориентированные именно на форматирование. В частности, это диалог параметров шрифта, представленный компонентом FontDialog.

Основным свойством этого диалога можно назвать свойство Font, при помощи которого данный диалог обменивается информацией о шрифте с программой. С типом TFont мы уже отчасти знакомы, в частности, с его свойствами Color, Name, Size, и Style. Так же имеются свойства Charset и Height, определяющие набор символов и высоту, соответственно. При этом значение высоты напрямую зависит от размера шрифта, задаваемого свойством Size. Вместе с тем, для диалога параметров шрифта

важно именно свойство Size, поскольку в самом диалоге пользователь указывает именно это свойство. Что касается свойства Charset, то его возможные значения определяются выбранным шрифтом. Для шрифтов OpenType, основанных на Unicode и используемых в новейших версиях Windows, допустимо наличие множества наборов символов, для обычных же TrueType шрифтов наборов бывает не более 2, обычно это основной латинский (127 символов ANSI) и один из дополнительных – греческий, восточноевропейский, кириллический и т.д. Значение набора символов задается при помощи констант, определенных в модуле Graphics, или непосредственно числами. Так, для ANSI это будет 0, для символьных шрифтов – 2, а для кириллических – 204.

Подобно другим диалогам, диалог параметров шрифта имеет свойство Options. В данном случае оно имеет следующий набор флагов:

- fdAnsiOnly – Делает доступными для выбора только шрифты, поддерживающие ANSI-символы. Символьные шрифты будут исключены;
- fdApplyButton – Отображает кнопку «Применить», действие для которой можно задать при помощи обработчика события onApply;
- fdEffects – Отображает группу видоизменения (эффекты зачеркивания, подчеркивания и цвет шрифта);
- fdFixedPitchOnly – Делает доступными только моноширинные шрифты;
- fdForceFontExist – Выполняет проверку на существование шрифта, введенного пользователем и выводит сообщение об ошибке, если такового нет;
- fdLimitSize – Включает проверку на границы размеров шрифта, задаваемых при помощи свойств MaxFontSize и MinFontSize;
- fdNoFaceSel – Диалог появляется без предварительно выбранного шрифта;
- fdNoOEMFonts – Исключает OEM-шрифты из списка доступных шрифтов;
- fdScalableOnly – Исключает не масштабируемые (битовые, или Type 1) шрифты из списка;
- fdNoSimulations – Отображает только те варианты начертания, которые явно определены в файлах шрифта. Генерируемые наклонные и полужирные начертания не предлагаются;
- fdNoSizeSel – Диалог появляется без предварительно выбранного размера;
- fdNoStyleSel – Диалог появляется без предварительно выбранного стиля;
- fdNoVectorFonts – Исключает векторные («плоттерные») шрифты;
- fdShowHelp – Отображает кнопку справки на диалоге;
- fdTrueTypeOnly – Делает доступными только шрифты типа TrueType;
- fdWysiwyg – Отображает только шрифты, доступные для экрана и принтера (кроме TrueType).

Чаще всего диалог шрифта используют совместно с текстовым редактором. Допустим, если у нас имеется приложение, состоящее из многострочного редактора (ком-

понент Memo) и предназначенное для просмотра текстовых файлов. Как минимум, такое приложение будет иметь 2 визуальных компонента – редактор и кнопку для вызова диалога открытия файла, а так же 1 невизуальный, представляющий собой собственно диалог открытья файла. Мы можем предоставить пользователю возможность изменять шрифт, которым отображаются файлы в редакторе, для чего нам потребуется компонент FontDialog и кнопка для обращения к этому диалогу. Не помешает добавить еще одну кнопку – для выхода из программы. Таким образом, мы получим приложение, вид которого в Delphi IDE будет примерно таким, как показано на рис. 14.3.

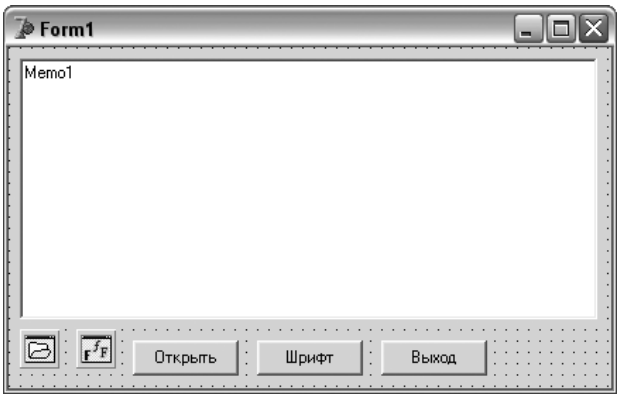

Рис. 14.3. Приложение для просмотра текстовых файлов

Из свойств, назначенных использованным компонентам, отметим лишь свойства Caption для всех 3 кнопок, а так же свойство Filter для диалога открытия файла. Пусть оно будет определено следующим образом:

```
Текстовые|*.txt;*.bat;*.ini;*.pas;*.dpr|Все|*.*
```

В данном случае фильтр будет иметь 2 варианта: «текстовые» (для файлов с расширениями txt, bat, ini, pas и dpr), а так же «все», для файлов с любым расширением. После этого остается добавить обработчики события OnClick для всех 3 кнопок, в результате чего мы получим программу, код которой приведен в листинге 14.1.

Листинг 14.1. Исходный код программы просмотра файлов с выбором шрифта

```
unit Unit1;
interface
uses
  Windows, Classes, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    OpenDialog1: TOpenDialog;
    FontDialog1: TFontDialog;
    procedure Button1Click(Sender: TObject);
```

```
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
var
  Form1: TForm1;
implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
begin
 if not OpenDialog1.Execute then exit;
 Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
 if not FontDialog1.Execute then exit;
 Memo1.Font:=FontDialog1.Font;
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
 close;
end;
end.
```

Как видно из процедуры, обрабатывающей нажатие кнопки «Шрифт», для того, чтобы изменить все параметры шрифта сразу, достаточно присвоить свойству Font компонента-редактора значение одноименного свойства диалога. Если бы нам требовалось присвоить лишь часть атрибутов (например, только название гарнитуры и размер кегля), то следовало бы присваивать соответствующие свойства типа TFont по отдельности:

```
Memo1.Font.Name:=FontDialog1.Font.Name;
Memo1.Font.Size:=FontDialog1.Font.Size;
```

Точно так же можно производить и обратную операцию, т.е. если нам надо добиться того, чтобы при вызове диалога все свойства были установлены в соответствие с теми, что имеются у редактора, то перед обращением к его методу Execute, было бы достаточно присвоить свойству Font диалога значение свойства Font редактора.

Диалог цвета

Диалог параметров шрифта позволяет установить любые его параметры, включая цвет. Но он не предназначен для того, чтобы изменять цвет заднего плана. Для этих целей следует использовать специальный диалог выбора цвета, в VCL он представлен компонентом ColorDialog. С точки зрения свойств он аналогичен диалогу параметров, с той лишь разницей, что вместо свойства Font, определяющем шрифт, у него имеется свойство Color, определяющее цвета.

Что касается типичного для диалогов свойства Options, то оно у ColorDialog имеет всего 5 флагов:

- cdFullOpen – Отображает диалог в развернутом виде, с палитрой выбора произвольных цветов;

- cdPreventFullOpen – Запрещает пользователю раскрывать диалог для выбора произвольных цветов;
- cdShowHelp – Отображает кнопку справки на диалоге;
- cdSolidColor – Указывает ОС на необходимость использования ближайшего системного цвета (из фиксированной палитры Windows) взамен выбранного пользователем:
- cdAnyColor – Позволяет пользователю выбрать цвет, который может быть отображен только путем смешивания нескольких цветов (что актуально, если глубина цветопередачи меньше 24 бит).

По умолчанию предлагается палитра, состоящая всего лишь из 48 цветов, кроме того, еще 16 цветов могут быть заданы разработчиком при помощи свойства CustomColors. Это свойство имеет тип TStrings и должно состоять из строк типа имя=значение, где в качестве имени используется слово Color и буква, от a до p по алфавиту (итого 16 вариантов максимум), а в качестве значения – цвет, заданный при помощи RGB-триплета:

```
ColorA=00CCDD
ColorB=DFCA72
...
ColorP=234567
```

Если же пользователь раскроет диалог (если это не запрещено флагом cdPreventFullOpen), или же диалог изначально открывается в полном виде (флаг cdFullOpen), то либо визуально, используя палитру, либо указывая числовые значения в формате HSB или RGB, пользователь сможет выбрать любой цвет (рис. 14.4).

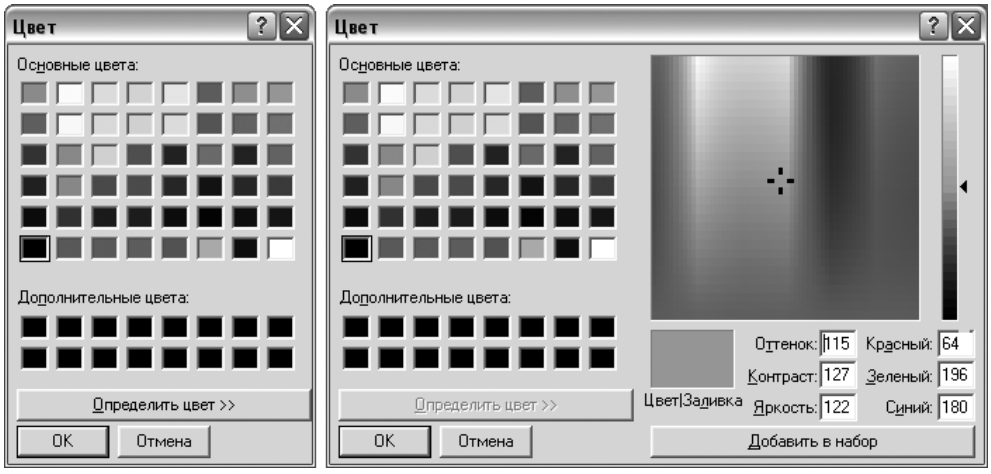

Рис. 14.4. Диалог выбора цвета в свернутом (слева) и в развернутом (справа) режиме

Чтобы проиллюстрировать работу этого диалога, добавим к приведенному в листинге 14.1 приложению компонент ColorDialog и еще одну кнопку, которую назовем «Фон». При этом код для события OnClick получится следующим:

```
procedure TForm1.Button4Click(Sender: TObject);
```

```
begin
 ColorDialog1.Color:=Memo1.Color;
 if not ColorDialog1.Execute then exit;
 Memo1.Color:=ColorDialog1.Color;
end;
```

Здесь вначале диалогу присваивается цвет фона редактора, после чего диалог вызывается, и, в случае положительного срабатывания, фону редактора назначается новый цвет. Модернизированное таким образом приложение можно посмотреть в каталоге Demo\Part3\Dlg2.

Редактор RTF

Рассмотренные выше диалоги для работы со шрифтами и с цветом представляют наибольшую ценность для форматирования текста. Для этих целей в Windows имеется специальный компонент – редактор форматированного текста, представленный в библиотеке VCL компонентом RichEdit, относящимся к компонентам Win32. Этот компонент фактически делает то же, что и все остальные компоненты этой группы – предоставляет простой и удобный доступ к одному из стандартных системных элементов управления, причем в данном случае таковой представлен отдельным системным файлом – richedit32.dll.

В иерархии классов VCL компонент RichEdit является наследником класса TCustomMemo, на основе которого построен «обычный» многострочный редактор. Вместе с тем, этот компонент обладает рядом свойств и методов, позволяющих, с одной стороны, форматировать текст, а с другой – обеспечивающих ряд сервисных функций, вроде поиска совпадений или вывода на печать. Всего в распоряжении редактора форматированного текста имеется свыше 100 свойств. Впрочем, непосредственно у класса TCustomRichEdit, на основе которого и создан рассматриваемый компонент, определено только 12, при этом ряд из них – HideSelection, Lines, SelLength, SelStart и SelText являются лишь переопределение одноименных свойств уже рассмотренных нами разновидностей редакторов. В итоге нам остается рассмотреть не так уж и много новых свойств этого компонента – см. таблицу 14.3.

Таблица 14.3. Собственные свойства редактора RTF

Свойство	Тип	Описание
DefAttributes	TTextAttributes	Определяет характеристики текста по умолчанию
DefaultConverter	TConversionClass	Определяет класс того объекта, который будет использоваться для преобразования формата текста. Автоматически используются преобразователи простого текста из и в RTF
HideScrollBars	Boolean	Определяет, должны ли полосы прокрутки появляться только при необходимости
PageRect	TRect	Определяет размеры страницы (в пикселях), которые будут использоваться для вывода на печать
Paragraph	TParaAttributes	Определяет параметры форматирования абзаца
PlainText	Boolean	Определяет тип текста – форматированный (false) или простой (true)
SelAttributes	TTextAttributes	Определяет характеристики текста выделенного фрагмента

Свойство DefAttributes типа TTextAttributes, задающее параметры шрифта, по своей сути похоже на свойство Font, имеющееся у многих других компонент, но доступно только во время выполнения. В то же время, в момент инициализации приложения, именно на основе параметров, указанных для свойства Font, и назначаются значения для свойства DefAttributes. Что касается типа TTextAttributes, то он имеет одно важное отличие от типа TFont, а именно – свойство ConsistentAttributes, которое показывает отличия выбранного фрагмента текста по отношению к остальному тексту. Впрочем, такая информация является более полезной для другого свойства редактора, а именно – для SelAttributes. Именно это свойство позволяет изменять параметры выделенной части текста, или же той его части, где находится каретка ввода. Поскольку текст может быть выделенным только в работающем приложении, то и это свойство доступно только во время выполнения.

Чтобы лучше представить себе суть свойств DefAttributes и SelAttributes, создадим небольшое приложение, которое выводило бы информацию о состоянии обоих этих свойств для компонента RichEdit. Для этого, помимо самого компонента RichEdit, нам понадобятся так же 2 компонента типа Memo – для вывода информации, а так же кнопка, по нажатию на которую интересующие нас сведения будут выводиться.

Кроме того, нам понадобится готовый файл в формате RTF, который будет содержать предварительно отформатированный различными способами текст. Подготовить его можно в любом текстовом процессоре, включая Word или WordPad, надо только будет при сохранении указать соответствующий формат файла. Для загрузки такого файла разместим на форме еще одну кнопку. В результате у нас получится форма с редактором RTF, 2 кнопками и 2 обычными редакторами (рис. 14.5).

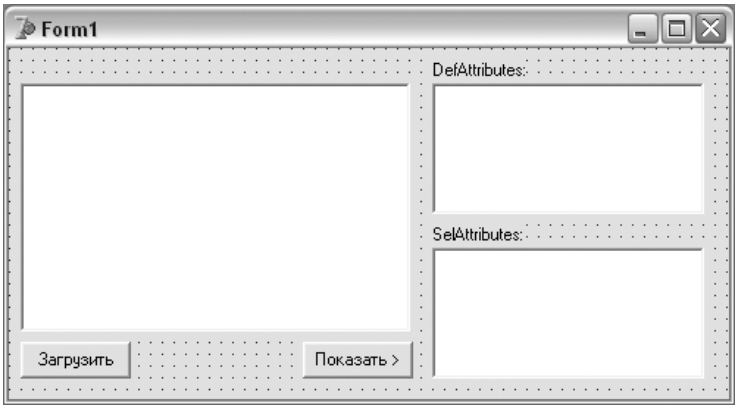

Рис. 14.5. Приложение для тестирования DefAttributes и SelAttributes

Для обработчика события OnClick кнопки «Загрузить» достаточно будет написать следующий код:

```
RichEdit1.Lines.LoadFromFile('simple.rtf');
```

Здесь подразумевается, что файл с примером текста называется simple.rtf и расположен в том же каталоге, что и исполняемый файл приложения.

Что касается кода, выводящего информацию о значениях исследуемых свойств, то, учитывая, что нам оба раза придется выводить сведения об объекте одного и того же типа – TtextAttribuitrs, то имеет смысл предварительно создать процедуру, которая

могла бы получить нужные данные на обработку и вывести их в указанное место. Назовем ее PrintAttrInfo, и определим как публичную процедуру класса TForm1:

```
type
  TForm1 = class(TForm)
    ...
  public
    procedure PrintAttrInfo(a: TTextAttributes; m: TMemo);
  end;
```

Реализация же этой процедуры будет заключаться в последовательном добавлении в указанный в качестве 2-го аргумента редактор строк со значениями атрибутов:

```
procedure TForm1.PrintAttrInfo(a: TTextAttributes; m: TMemo);
var
 s: string;
begin
 m.Lines.Clear; //предварительно очищаем содержимое
 m.Lines.Add('Charset: '+IntToStr(a.Charset)); //набор символов
 m.Lines.Add('Colour: '+IntToStr(a.Color)); //цвет
 m.Lines.Add('Name: '+a.Name); //гарнитура
{ для текстового вывода информации о типе шрифта определим, какой из 3
возможных вариантов используется }
 case a.Pitch of
   fpDefault: m.Lines.Add('Pitch: fpDefault');
   fpVariable: m.Lines.Add('Pitch: fpVariable');
   fpFixed: m.Lines.Add('Pitch: fpFixed');
 end;
 m.Lines.Add('Size: '+IntToStr(a.Size)); //размер
{ поскольку шрифт может одновременно иметь сразу несколько признаков
свойства Style, то проверим их все }
 if fsBold in a.Style then s:='fsBold ';
 if fsItalic in a.Style then s:=s+'fsItalic ';
 if fsUnderline in a.Style then s:=s+'fsUnderline ';
 if fsStrikeOut in a.Style then s:=s+'fsStrikeOut ';
 m.Lines.Add('Style: [ '+s+']');
end;
```

Наконец, для кнопки «Показать» остается написать 2 вызова определенной нами процедуры PrintAttrInfo:

```
PrintAttrInfo(RichEdit1.DefAttributes, Memo1);
PrintAttrInfo(RichEdit1.SelAttributes, Memo2);
```

Если теперь запустить это приложение и нажать на кнопку «Показать», то можно будет увидеть, что для обоих свойств отображаются одни и те же значения. Если же загрузить файл с форматированным текстом, то значения свойства SelAttributes изменятся. Более того, если в загруженном файле применены различные шрифты или стили оформления, то, изменяя текущую позицию каретки, можно будет видеть текущие атрибуты шрифта. В то же время, если изменить значение свойства Font, то изменятся не только значение свойства DefAttributes, но и параметры всего текста. Чтобы в этом убедиться, можно добавить еще одну кнопку, которая будет вызывать компонент FontDialog и написать для нее следующий код:

```
FontDialog1.Font:=RichEdit1.Font;
if FontDialog1.Execute then RichEdit1.Font:=FontDialog1.Font;
```

Таким образом, после обращения к диалогу шрифта и назначению новых данных свойству Font, изменится оформление всего текста. Готовый пример можно найти в каталоге Demo\Part3\Rich1.

Еще одно свойство, специфическое для редактора RTF – это Paragraph. Оно позволяет задать ряд параметров текста, относящихся к форматированию абзацев, включая выравнивание, стиль списка, размер отступов и табуляции. Оно имеет тип TParaAttributes, который, в свою очередь, содержит следующие свойства:

- Alignment – определяет выравнивание, может принимать значения taLeftJustify, taRightJustify, taCenter для выравнивания по левому краю, по правому краю и по центру, соответственно;
- FirstIndent – определяет размер «красной строки» в пикселях;
- LeftIndent и RightIndent – определяют, соответственно, отступы от левого и правого полей в пикселях;
- Numbering – отвечает за стиль «нумерации», должен ли параграф оформляется как пункт списка (nsBullet) или нет (nsNone).

При работе с редактором форматированного текста следует учитывать, что в разных версиях Windows поддерживаются разные версии редактора. Так, в Windows 95 это версия 1.0, в Windows 98 – 2.0, а в Windows 2000 и XP – 3.0. Вместе с тем, если в системе с Windows 95 установлен Office 97 или MSIE 4.0, то компонент будет обновлен до версии 2.0. Разумеется, новые версии имеют обратную совместимость с предыдущими, однако на практике программа, использующая этот компонент, и запущенная под Windows 95 будет вести себя не совсем так, как та же программа, работающая в Windows XP. В основном, это связано с ошибками, допущенными при разработке первой редакции этого элемента управления для Windows 95. И хотя вряд ли вы найдете сейчас компьютер, работающий под этой версией ОС, да еще и без установленного Office, требования к обратной совместимости заставляют Borland от версии к версии Delphi оставлять свой компонент RichEdit привязанным к версии 1.0 данного элемента Windows. Поэтому, если вам придется на практике разрабатывать приложение, построенное вокруг этого компонента (т.е. приложение, в котором текстовый редактор является одной из важнейших составляющих), то используете сторонние компоненты, ориентированные на более новые версии этого системного элемента, например, RichEdit98. В частности, помимо более предсказуемого поведения и отсутствия специфических «особенностей», новые версии имеют более широкую функциональность, например, возможность выравнивания текста по ширине, а так же куда большие возможности оформления. В то же время, если RichEdit нужен лишь как обычный блокнот с готовыми функциями поиска и печати, то функциональности стандартного компонента будет вполне достаточно.

Пример текстового редактора

Все рассмотренные нами стандартные диалоги, совместно с компонентом RTF, могут быть использованы в рамках одного приложения – текстового редактора. Поэтому для того, чтобы лучше разобраться с их использованием, а заодно задействовать ряд иных, рассмотренных ранее в этой части книги компонент, создадим приложение,

являющееся простым текстовым редактором – немного упрощенным аналогом входящего в состав Windows редактора WordPad.

Но прежде, чем браться за разработку приложения, хотелось бы остановиться на таком аспекте, как правила именования компонент. По умолчанию, как мы знаем, Delphi присваивает им имена, убирая первую букву (T) и добавляя порядковый номер (1,2,3). Но на практике это не очень удобно, поскольку в достаточно большой программе трудно будет понять, что такое Button22 или CheckBox17. По этой причине существуют некие общие правила наименования компонентов. Сводятся они к тому, что используется 2-3-4 буквенный префикс (или, по желанию – суффикс), идентифицирующий принадлежность объекта к классу, в сочетании со словом, отражающем суть данного объекта. Например, текстовое поле (класс TEdit), предназначенное для ввода имени пользователя, согласно таким правилам может называться, скажем, EdtUserName или UserNameEd, а кнопка (TButton) запуска чего-либо – BtnStart или StartBtn.

Итак, для текстового редактора нам понадобится создать новое приложение, после чего разместить на его форме ряд компонент, необходимых для его работы. Прежде всего, это главное меню (MainMenu) и, разумеется, сам RichEdit. Очевидно, что нам пригодятся и стандартные диалоги, включая диалоги для работы с файлами (OpenDialog и CloseDialog), диалоги печати (PrintDialog и PrintSetupDialog), а так же диалоги для работы с цветом и шрифтами (FontDialog и ColorDialog). Теперь для компонента RichEdit установим свойство Align в AlClient, чтобы область редактора заняла все свободное место на форме (рис. 14.6).

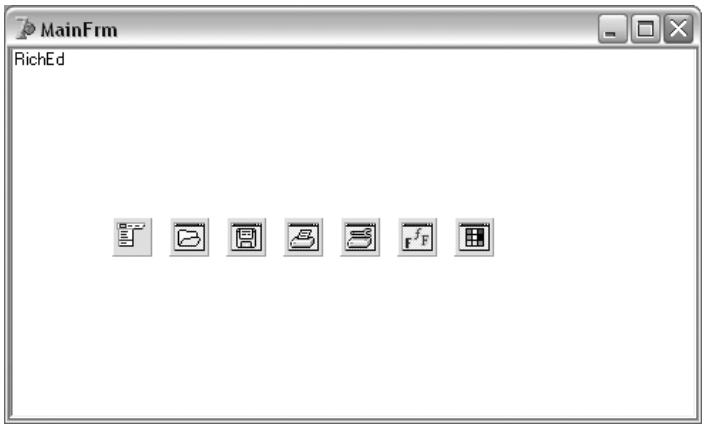

Рис. 14.6. Окно редактора в начале разработки

Разместив нужные нам компоненты на форме, установим имена следующим образом:

- Форма – MainFrm;
- Главное меню – MainMenu;
- Редактор – RichEd;
- Диалоги – OpenDlg, SaveDlg, PrintDlg, PrintSetupDlg, FontDlg и ColorDlg;

Теперь можно заняться обустройством главного меню. Предусмотрим в нем 3 раздела: Файл, Правка и Формат. Для этого откроем редактор меню (двойным щелчком по компоненту MainMenu) и создадим эти 3 основных пункта главного меню. Они автоматически получат названия N1, N2 и N3, но изменять их мы не будем по той простой причине, что эти пункты – лишь заголовки для самих меню, и в программном коде обращаться к ним нам не придется.

Далее заполним меню «Файл», создав в нем пункты «Открыть…», «Сохранить…», «Печать…», «Принтер…» и «Выход». Назовем их OpenFileM, SaveFileM, PrintFileM, PrintSetupFileM и ExitFileM. Общие суффиксы FileM в будущем, будут нам подсказывать, что мы имеем дело с меню файловых операций. А многоточия после слова в подписи первых 4 пунктов меню будут подсказывать пользователю, что данные пункты выполняются не сами по себе, а вызывают диалоговое окно. Внешний вид конструктора меню на данном этапе будет таким, как показано на рис. 14.7. Само меню в программе будет выглядеть практически так же, что может навести нас на мысль о необходимости добавить разделители перед пунктами «Печать» и «Выход». Для этого следует выбрать в конструкторе нужный пункт и нажать клавишу Ins на клавиатуре, после чего в появившемся новом пункте в качестве значения свойства Caption указать символ «-».

Следующим пунктом у нас идет меню «Правка». Разместим в нем стандартные пункты «Отменить», «Вырезать», «Копировать», «Вставить» и «Выделить все», присвоив им имена UndoEdM, CutEdM, CopyEdM, InsertEdM и SelAllEdM, соответственно. При этом после отмены не помешает поместить разделитель. Наконец, в последнюю группу – «Формат» – внесем пункты «Шрифт…» и «Цвет…», назвав их FontFmtM и ColorFmtM.

Теперь подготовим к работе файловые диалоги. Поскольку мы имеем дело с текстовым редактором типа RTF, то основным типом файла будет как раз RTF. Тем не менее, не помешает предусмотреть возможность открытия файлов другого типа, в частности обычных текстовых (TXT). Таким образом, для свойства Filter в инспекторе объекта следует указать следующее значение:

```
Файлы RTF|*.rtf|Текстовые файлы ASCII|*.txt|Все файлы|*.*
```

Что касается свойства DefaultExt, то для диалога сохранения было бы целесообразным указать rtf.

Для начала подготовительной работы уже сделано достаточно, впору приступать к разработке собственно рабочих функций программы. В частности, для пункта «Открыть» из меню «Файл» для события OnClick достаточно написать следующий код:

```
if OpenDlg.Execute then RichEd.Lines.LoadFromFile(OpenDlg.FileName);
```

Для сохранения было бы целесообразным попытаться заранее назначить файлу имя, соответствующее имени файла в диалоге открытия:

```
if if OpenDlg.FileName<>'' then SaveDlg.FileName:=OpenDlg.FileName;
if SaveDlg.Execute then RichEd.Lines.SaveToFile(SaveDlg.FileName);
```

Подобным образом выполняются и обращения к остальным диалогам в программе. Что касается элементов меню «Правка», то там так же не должно быть проблем: достаточно обращаться к соответствующим методам самого редактора для выполнения соответствующих операций. При этом, правда, было бы полезным предварительно

проверять операцию на допустимость. Например, перед выполнением отмены (метод Undo) не помешает проверить состояние свойства-флага CanUndo:

```
if RichEd.CanUndo then RichEd.Undo;
```

То же самое относится и к операциям работы с буфером обмена – не к чему пытаться копировать в буфер выделенный текст, если такового нет:

```
if RichEd.SelText<>'' then RichEd.CutToClipboard;
```

Пояснить следует, разве что, операцию вставки из буфера обмена: здесь было бы полезным предварительно убедиться, что в нем действительно находится текст. Для этого в список используемых модулей (uses) следует добавить модуль Clipbrd, сама же процедура проверки будет выглядеть следующим образом:

```
if Clipboard.HasFormat(CF_TEXT) then ...
```

Здесь мы обратились к глобальному объекту Clipboard (он создается автоматически, подобно Screen или Application) и воспользовались его методом HasFormat, чтобы убедиться, что информация, находящаяся в буфере обмена, является текстом.

После создания обработчиков событий для всех пунктов меню, остается сохранить проект, выбрав в качестве имени файла формы «main», а файла проекта – «myedit». Таким образом, полный исходный код модуля main получится примерно таким, как показано в листинге 14.2.

Листинг 14.2. Исходный код редактора MyEdit

```
unit main;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, StdCtrls, ComCtrls, ToolWin, Clipbrd;
type
  TMainFrm = class(TForm)
    MainMenu: TMainMenu;
    ToolBar: TToolBar;
    RichEd: TRichEdit;
    OpenDlg: TOpenDialog;
    SaveDlg: TSaveDialog;
    PrintDlg: TPrintDialog;
    PrintSetupDlg: TPrinterSetupDialog;
    FontDlg: TFontDialog;
    ColorDlg: TColorDialog;
    N1: TMenuItem;
    N2: TMenuItem;
    N3: TMenuItem;
    OpenFileM: TMenuItem;
    SaveFileM: TMenuItem;
    PrintM: TMenuItem;
    PrintSetupM: TMenuItem;
    ExitFileM: TMenuItem;
    N4: TMenuItem;
    N5: TMenuItem;
```

```
    UndoEdM: TMenuItem;
    N6: TMenuItem;
    CutEdM: TMenuItem;
    CopyEdM: TMenuItem;
    InsertEdM: TMenuItem;
    SelAllEdM: TMenuItem;
    FontFmtM: TMenuItem;
    ColorFmtM: TMenuItem;
    procedure OpenFileMClick(Sender: TObject);
    procedure SaveFileMClick(Sender: TObject);
    procedure PrintMClick(Sender: TObject);
    procedure PrintSetupMClick(Sender: TObject);
    procedure ExitFileMClick(Sender: TObject);
    procedure FontFmtMClick(Sender: TObject);
    procedure ColorFmtMClick(Sender: TObject);
    procedure UndoEdMClick(Sender: TObject);
    procedure CutEdMClick(Sender: TObject);
    procedure CopyEdMClick(Sender: TObject);
    procedure InsertEdMClick(Sender: TObject);
    procedure SelAllEdMClick(Sender: TObject);
  end;
var
  MainFrm: TMainFrm;
implementation
{$R *.dfm}
procedure TMainFrm.OpenFileMClick(Sender: TObject);
begin
 if OpenDlg.Execute then RichEd.Lines.LoadFromFile(OpenDlg.FileName);
end;
procedure TMainFrm.SaveFileMClick(Sender: TObject);
begin
 if if OpenDlg.FileName<>'' then SaveDlg.FileName:=OpenDlg.FileName;
 if SaveDlg.Execute then RichEd.Lines.SaveToFile(SaveDlg.FileName);
end;
procedure TMainFrm.PrintMClick(Sender: TObject);
begin
 if PrintDlg.Execute then RichEd.Print('');
end;
procedure TMainFrm.PrintSetupMClick(Sender: TObject);
begin
 PrintSetupDlg.Execute;
end;
procedure TMainFrm.ExitFileMClick(Sender: TObject);
begin
 close;
end;
procedure TMainFrm.FontFmtMClick(Sender: TObject);
```

```
begin
 FontDlg.Font.Name:=RichEd.SelAttributes.Name;
 FontDlg.Font.Color:=RichEd.SelAttributes.Color;
 FontDlg.Font.Charset:=RichEd.SelAttributes.Charset;
 FontDlg.Font.Size:=RichEd.SelAttributes.Size;
 FontDlg.Font.Style:=RichEd.SelAttributes.Style;
 if not FontDlg.Execute then exit;
 RichEd.SelAttributes.Name:=FontDlg.Font.Name;
 RichEd.SelAttributes.Color:=FontDlg.Font.Color;
 RichEd.SelAttributes.Charset:=FontDlg.Font.Charset;
 RichEd.SelAttributes.Size:=FontDlg.Font.Size;
 RichEd.SelAttributes.Style:=FontDlg.Font.Style;
end;
procedure TMainFrm.ColorFmtMClick(Sender: TObject);
begin
 ColorDlg.Color:=RichEd.SelAttributes.Color;
 if not ColorDlg.Execute then exit;
 RichEd.SelAttributes.Color:=ColorDlg.Color;
end;
procedure TMainFrm.UndoEdMClick(Sender: TObject);
begin
 if RichEd.CanUndo then RichEd.Undo;
end;
procedure TMainFrm.CutEdMClick(Sender: TObject);
begin
 if RichEd.SelText<>'' then RichEd.CutToClipboard;
end;
procedure TMainFrm.CopyEdMClick(Sender: TObject);
begin
 if RichEd.SelText<>'' then RichEd.CopyToClipboard;
end;
procedure TMainFrm.InsertEdMClick(Sender: TObject);
begin
 if Clipboard.HasFormat(CF_TEXT) then RichEd.PasteFromClipboard;
end;
procedure TMainFrm.SelAllEdMClick(Sender: TObject);
begin
 RichEd.SelectAll;
end;
end.
```

Если теперь запустить программу (готовый исходный код находится в каталоге Demo\Part3\Editor), то можно будет убедиться, что она действительно работает - открывает файлы, позволяет вводить и редактировать текст, изменяя его оформление. Из недостатков можно сходу выделить лишь надпись «RichEd», которую редактор содержит изначально и отсутствие полос прокрутки, если текста будет больше, чем помещается в окне. Первый недостаток исправляется правкой свойства Lines, путем удаления ненужного текста, а второй – путем назначения свойству ScrollBars значе-

ния ssBoth. Отметим, что если при этом свойство HideScrollBars оставить в значении истины, то полосы прокрутки будут появляться лишь при необходимости.

Более серьезный недостаток кроется в заложенной нами возможности работать не только с файлами RTF, но и с обычными текстовыми. Дело в том, что если при сохранении вы даже выберите тип текстовых файлов и укажете расширение txt, то программа все равно сохранит файл с разметкой. Но, как нам известно, у компонента RichEdit существует свойство PlainText, отвечающее за текущий формат. Таким образом, остается лишь установить нужное значение для этого свойства в момент перед сохранением файла. А для того, чтобы определить, какой формат выбрал пользователь, используем свойство FilterIndex диалога сохранения:

```
if SaveDlg.Execute then begin
  RichEd.PlainText:=(SaveDlg.FilterIndex>1);
  RichEd.Lines.SaveToFile(SaveDlg.FileName);
  RichEd.PlainText:=false;
end;
```

Отметим, что после сохранения выполняется принудительная установка формата RTF. Это необходимо с той точки зрения, что следующим шагом пользователя может быть попытка открытия файла RTF, который не сможет быть обработан, если включен режим PlainText.

Диалоги поиска и замены

Работа с текстом не ограничивается изменением параметров шрифта. Часто гораздо более полезными операциями являются такие, как поиск подстроки с возможной заменой ее на иной текст. Для этих целей так же предусмотрены стандартные диалоговые окна – FindDialog и ReplaceDialog. Свойства этих компонентов включают в себя стандартное для диалогов Options, а так же группу свойств, определяющих положение диалога на экране – Left, Top и Position. Но наиболее важным, безусловно, является свойство FindText, которое собственно и содержит строку для поиска. У диалога замены предусмотрено еще одно свойство – ReplaceText, которое определяет строку для замены.

Но рассмотрим для начала свойство Options. Применительно к диалогам поиска и замены оно имеет следующий набор флагов:

- frDisableMatchCase – Делает недоступной опцию «С учетом регистра»;
- frDisableUpDown – Делает недоступной опцию выбора направления поиска;
- frDisableWholeWord - Делает недоступной опцию «Только слово целиком»;
- frDown – Делает включенной опцию направления поиска «Вниз» (если этот флаг выключен, то будет выбрано направление «Вверх»);
- frFindNext – Этот флаг включается автоматически, когда пользователь щелкает по кнопке «Найти далее»;
- frHideMatchCase – Удаляет опцию «С учетом регистра»;
- frHideWholeWord – Удаляет опцию «Только слово целиком»;
- frHideUpDown – Удаляет опцию выбора направления поиска;

- frMatchCase – Указывает, что выбрана опция «С учетом регистра»;
- frReplace – Указывает, что должна быть произведена замена данного найденного вхождения (только для диалога замены);
- frReplaceAll - Указывает, что должна быть произведена замена всех найденных вхождений (только для диалога замены);
- frShowHelp – Отображает кнопку справки на диалоге;
- frWholeWord – Указывает, что была выбрана опция «Только слово целиком».

По умолчанию включен только флаг frDown, что делает выбранным направление поиска к концу документа.

Использование данных диалогов подразумевает применение самостоятельно разработанных функций поиска и замены. В то же время, у такого компонента, как редактор RTF, имеется метод FindText, который существенно упрощает задачу. Он определен следующим образом:

```
function FindText(const SearchStr: string; StartPos, Length: Integer;
Options: TSearchTypes): Integer;
```

Здесь в качестве SearchStr указывается строка для поиска, StartPos обозначает место, с которого следует начинать поиск, а Length – место, до которого следует производить поиск. В качестве Options можно указать флаги stWholeWord и stMatchCase, включающие распознавание слов целиком и регистра. Таким образом, мы можем модернизировать наш текстовый редактор таким образом, чтобы он поддерживал поиск текста (модернизированный вариант находится в каталоге Demo\Part3\Editor2).

Прежде всего, поместим на форму оба рассматриваемых компонента – FindDialog и ReplaceDialog, присвоив им имена FindDlg и ReplaceDlg. Затем откроем конструктор меню и в раздел «Правка» добавим разделитель и 2 новых пункта – «Найти…» и «Заменить...», назвав их, соответственно, SearchEdM и ReplaceEdM. Теперь для пункта «Найти» определим процедуру вызова диалога поиска:

```
procedure TMainFrm.SearchEdMClick(Sender: TObject);

begin

 FindDlg.Execute;

end;
```

Таким способом мы лишь отображаем диалог. Саму процедуру поиска следует разместить в обработчике события OnFind самого диалога. В простейшем случае она должна лишь определять место начала и конца поиска, а в случае нахождения искомого – выделять найденный фрагмент. Таким образом, мы можем получить примерно следующую процедуру:

```
procedure TMainFrm.FindDlgFind(Sender: TObject);
var
 StartPos, ToPos, FoundPos: Integer;
begin
 StartPos:=RichEd.SelStart+RichEd.SelLength;
 ToPos:=Length(RichEd.Text)-StartPos;
```

```
 FoundPos:=RichEd.FindText(FindDlg.FindText, StartPos, ToPos, []);
 if FoundPos<>-1 then begin
   RichEd.SelStart:=FoundPos;
   RichEd.SelLength:=Length(FindDlg.FindText);
   RichEd.SetFocus;
 end else ShowMessage('Текст не найден!');
end;
```

Здесь вначале вычисляется точка начала поиска – StartPos, затем определяется длина поиска – ToPos, в типичном случае она должна равняться размеру текста от точки начала поиска до конца документа, после чего вызывается собственно метод FindText, а результат его работы присваивается переменной FoundPos. Затем, если результат не равен -1 (т.е. если вхождение подстроки найдено), фрагмент текста выделяется, и фокус ввода передается окну редактора, в противном случае выводится сообщение «Текст не найден».

Недостатком процедуры в таком виде является то, что она не учитывает возможных изменений, сделанных пользователем в окне поиска. В частности, не определяется ни варианты учета регистра символов, ни вариант поиска по словам. Для того чтобы задействовать эти опции, нам необходимо определить переменную типа TSearchTypes, которую надо будет установить в то или иное значение, в зависимости от состояния флагов диалога поиска. Инициализацию этой переменной (назовем ее Opt) следует проводить перед обращением к методу FindText, так что в начало процедуры добавим следующие строки кода:

```
if frMatchCase in FindDlg.Options then Opt:=[stMatchCase];
if frWholeWord in FindDlg.Options then Opt:=Opt+[stWholeWord];
```

Теперь наша функция поиска распознает заданные пользователем установки, так что можно перейти к реализации процедуры замены. Собственно вызов диалога замены производится точно таким же способом – путем обращения к методу Execute. При этом процедура поиска, реализуемая диалогом замены, будет в точности такой же, как и у диалога поиска – достаточно будет изменить имя компонента (см. так же листинг 14.3).

Что касается процедуры обработки события OnReplace, которое возникает, когда пользователь нажимает на кнопку «Заменить» или «Заменить все», то она, в простейшем случае будет выглядеть аналогичным образом, с той лишь разницей, что к ней добавляется замена вхождения на указанный пользователем текст. А именно после выделения найденного текста следует добавить еще одну строку кода:

```
RichEd.SelText:=ReplaceDlg.ReplaceText;
```

Таким образом, одиночную замену можно считать реализованной. Однако если пользователь нажмет кнопку «Заменить все», то произойдет лишь одна замена. Следовательно, необходимо использовать цикл с постусловием. При этом цикл должен прерываться после первой же итерации, если пользователь нажал кнопку «Заменить», а не «Заменить все», либо в том случае, если искомой строки не найдено. Таким образом, мы получаем следующее условие:

```
repeat
...
until (FoundPos=-1) or not(frReplaceAll in ReplaceDlg.Options);
```

Если теперь заключить весь блок операций процедуры в этот цикл, за исключением, разве что, первых 2 строк, которые определяют параметры поиска, запустить программу и попытаться произвести замену по всем вхождениям, то можно будет убедиться, что все замечательно работает. Единственная проблема состоит в том, что по окончании работы, вне зависимости от того, были ли произведены замены или нет, вы получите сообщение «Текст не найден». Это объясняется тем, что при последней итерации в любом случае будет выполнен блок else условного оператора, проверяющего наличие вхождения искомого текста. Таким образом, необходимо вынести эту проверку за пределы цикла, а для того, чтобы определить, было ли что-либо найдено, используем переменную-счетчик, которую перед началом выполнения цикла установим в 0, а при каждой выполненной замене будем увеличивать на 1. Таким образом, мы сможем не только выдавать сообщение о том, что текст не найден, но и показывать количество произведенных замен, если пользователь нажимал на кнопку «Заменить все». Итоговый вариант процедуры замены приведен в листинге 14.3.

Листинг 14.3. Процедуры поиска и замены для редактора RTF и ReplaceDialog

```
procedure TMainFrm.ReplaceDlgFind(Sender: TObject);
var
 StartPos, ToPos, FoundPos: Integer;
 Opt: TSearchTypes;
begin
 if frMatchCase in ReplaceDlg.Options then Opt:=[stMatchCase];
 if frWholeWord in ReplaceDlg.Options then Opt:=Opt+[stWholeWord];
 StartPos:=RichEd.SelStart+RichEd.SelLength;
 ToPos:=Length(RichEd.Text)-StartPos;
 FoundPos:=RichEd.FindText(ReplaceDlg.FindText, StartPos, ToPos, Opt);
 if FoundPos<>-1 then begin
   RichEd.SelStart:=FoundPos;
   RichEd.SelLength:=Length(ReplaceDlg.FindText);
   RichEd.SetFocus;
 end else ShowMessage('Текст не найден!');
end;
procedure TMainFrm.ReplaceDlgReplace(Sender: TObject);
var
 i, StartPos, ToPos, FoundPos: Integer;
 Opt: TSearchTypes;
begin
 if frMatchCase in ReplaceDlg.Options then Opt:=[stMatchCase];
 if frWholeWord in ReplaceDlg.Options then Opt:=Opt+[stWholeWord];
 i:=0;
 repeat
   StartPos:=RichEd.SelStart+RichEd.SelLength;
   ToPos:=Length(RichEd.Text)-StartPos;
   FoundPos:=RichEd.FindText(ReplaceDlg.FindText, StartPos, ToPos, Opt);
   if FoundPos<>-1 then begin
     RichEd.SelStart:=FoundPos;
     RichEd.SelLength:=Length(ReplaceDlg.FindText);
     RichEd.SelText:=ReplaceDlg.ReplaceText;
```

```
      RichEd.SetFocus;
      inc(i);
    end;
 until (FoundPos=-1) or not(frReplaceAll in ReplaceDlg.Options);
 if i=0 then ShowMessage('Текст не найден!') else
   if frReplaceAll in ReplaceDlg.Options then
     ShowMessage('Произведено '+IntToStr(i)+' замен');
end;
```

Таким образом, мы создали редактор, который, в принципе, может делать все необходимые в повседневной работе вещи. Для удобства остается только определить сочетания горячих клавиш для основных действий, прежде всего – для открытия и сохранения файлов, а так же для поиска и замены (горячие клавиши для редактирования текста типа Ctrl+Z или Ctrl+C поддерживаются автоматически). Чтобы назначить сочетания, в конструкторе меню для меню «Открыть» в свойстве ShortCut выберем Ctrl+O, для «Сохранить» – Ctrl+S, для «Найти» – Ctrl+F, а для «Заменить» – Ctrl+H. Все эти сочетания являются стандартными для данных действий в Windows, а изобретать собственные варианты для стандартных действий крайне не рекомендуется.

Работа с файлами и мультимедиа

Продолжая знакомство с библиотекой VCL, рассмотрим еще несколько групп компонентов, в частности, для работы с файлами, а так же компоненты для работы с графикой и видео. Здесь следует отметить, что данные компоненты, в отличие от уже рассмотренных диалогов или коллекции изображений, являются визуальными. Также примечателен тот факт, что некоторые компоненты для работы с файлами, которые мы рассмотрим в этой главе, относятся к группе 16-разрядных элементов интерфейса, т.е. достались в наследство со времен Windows 3.1 и Delphi 1. Тем не менее, во многих случаях их использование бывает вполне уместно.

Списки файлов и каталогов

Самый простой путь отображения содержимого дисков ПК в Delphi – это использование компонентов FileListBox (список файлов) и DirectoryListBox (список каталогов). Оба этих компонента появились еще в 1-й версии Delphi и относятся к группе компонент Win 3.1. Компонент списка файлов позволяет просматривать содержимое указанного каталога. Он является наследником обычного списка (ListBox) и имеет набор дополнительных свойств, определяемых собственной спецификой. С ними можно ознакомиться в таблице 15.1.

Таблица 15.1. Свойства FileListBox

Свойство	Тип	Описание
Directory	String	Определяет каталог, файлы которого должны отображаться в данном списке
Drive	Char	Определяет букву диска, файлы которого должны отображаться в данном списке
FileEdit	TEdit	Связывает данный список с полем редактирования, в котором будет отображаться выбранный файл
FileName	String	Определяет имя текущего выбранного файла в списке (включая путь файла)

FileType	TFileType	Определяет, файлы какого типа (по атрибутам) должны отображаться в списке
Mask	String	Ограничивает видимые в списке файлы по маске
ShowGlyphs	Boolean	Указывает, должны ли отображаться иконки файлов, коих предусмотрено всего 2 типа - доя выполняемых и для остальных

Следует учитывать, что ряд свойств, в частности, Directory, Drive и FileName являются взаимозависимыми. Т.е., скажем, установив определенное значение для FileName, вы можете параллельно изменить Drive и Directory (например, если файл находится на другом диске). Свойства FileType и Mask позволяют ограничить отображаемые файлы, отбирая их по принципу принадлежности к той или иной группе по атрибутам (FileType) или по шаблону имени (Mask). Шаблон задается в соответствии с обычными для Windows правилами подстановки масок, т.е. с использованием подстановочных знаков «*» и «?». Что касается атрибутов, то для свойства FileType предусмотрены следующие флаги:

- ftReadOnly – Отображаются только файлы с атрибутом «только чтение»;
- ftHidden – Отображаются только файлы с атрибутом «скрытый»;
- ftSystem – Отображаются только файлы с атрибутом «системный»;
- ftVolumeID – Должна отображаться метка диска;
- ftDirectory – Будут отображаться каталоги;
- ftArchive – Отображаются только файлы, подлежащие архивации;
- ftNormal – Отображаются любые файлы без специальных атрибутов.

Следует учитывать, что если даже включен флаг ftDirectory и каталоги отображаются в списке, автоматическая смена каталога при щелчке пользователем мышкой, этим компонентом не предусмотрена.

В то же время, не будем забывать, что у нас имеется еще один компонент – DirectoryListBox, который как раз и предназначен для навигации по каталогам диска. Он имеет всего 4 собственных свойства – Directory, DirLabel, Drive и FileList. При этом свойства Drive и Directory полностью аналогичны таковым у FileListBox, а свойство DirLabel весьма похоже на FileEdit, с той лишь разницей, что если с FileListBox ассоциируется однострочный редактор, то с DirLabel – метка, свойство Caption которой и подлежит изменению в зависимости от выбранного каталога.

Последнее свойство DirectoryListBox – FileList. Оно служит для ассоциирования данного списка каталогов с компонентом – списком файлов. Имея связанные таким образом компоненты, мы получим автоматически работающую связку, в которой при изменении текущего каталога в списке каталогов, будет автоматически производиться смена каталога в ассоциированном списке файлов.

Рассмотрим эту пару компонент на простом примере. Для этого создадим приложение и поместим на его главную форму следующие компоненты, разместив их один под другим: Label, DirectoryListBox, Edit и FileListBox. Назовем их DirLbl, DirLst, FileEd и FileLst, после чего установим свойство DirLabel списка каталогов в DirLbl, его же свойство FileList – в FileLst, а свойство FileEdit списка файлов – в FileEd. Саму программу можно назвать FileView, или FV. Запустив приложение, можно будет убе-

диться, что в зависимости от того, какой каталог в списке каталогов выбран, меняется надпись метки и содержимое списка файлов, а при выборе файла его имя отображается в текстовом редакторе (рис. 15.1).

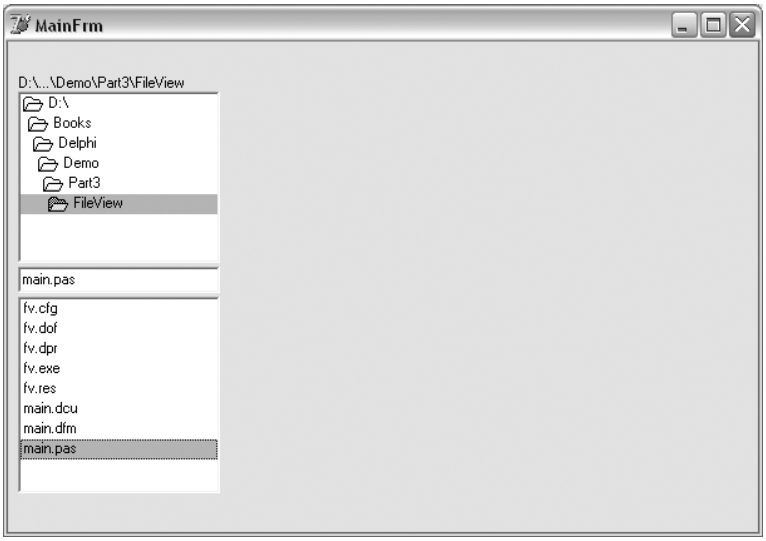

Рис. 15.1. Работа программы просмотра содержимого каталогов

Отдельно хочется отметить, что в данном случае мы не написали ни одной строчки кода, получив при этом, в общем-то, вполне работающее приложение.

Файловые компоненты – списки

В только что рассмотренном примере программы для просмотра файлов явно не хватает возможности выбора диска. Для этих целей предусмотрен отдельный компонент, DriveComboBox – список дисков, в котором в виде ниспадающего списка отображаются все дисковые устройства ПК. Нетрудно догадаться, что этот компонент основан на обычном комбинированном списке (ComboBox), а все его отличия заключаются в нескольких дополнительных свойствах, связанных со спецификой применения этого компонента. Всего таких свойств 3 – DirList, Drive и TextCase.

Свойство DirList предназначено для ассоциирования списка дисков со списком каталогов, а свойство Drive указывает или задает букву выбранного диска. Таким образом, если поместить на форму нашего приложения (FileView) этот компонент и установить его свойство DirList в значение DirLst, то мы получим уже полностью функциональную программу для просмотра содержимого всех дисков компьютера.

Что касается свойства TextCase, то оно определяет, в каком регистре должны выводиться метки дисков: если установлено принятое по умолчанию значение tcLowerCase, то метки дисков будут отображаться в нижнем регистре, а если tcUpperCase, то в верхнем.

Теперь нам остается рассмотреть последний компонент, относящийся к группе унаследованных элементов управления файлами. Это список фильтров – FilterComboBox, который обеспечивает возможность быстрой установки фильтра (маски) для списка

файлов. Подобно списку дисков, список фильтров основан на комбинированном списке, и так же имеет 3 собственных свойства, а именно FileList, Filter и Mask. Очевидно, что свойство FileList служит для ассоциирования этого компонента со списком файлов, а свойство Filter определяет сами фильтры. При этом синтаксис для определения фильтров у FilterComboBox полностью соответствует таковому у файловых диалогов. Наконец, свойство Mask позволяет узнать, какой именно фильтр используется в данный момент.

Таким образом, чтобы посмотреть, как взаимодействует «полный комплект» компонент управления файлами, поместим в самый низ формы список фильтров, и для свойства FileList укажем FileLst. Теперь можно определить сами фильтры: по умолчанию предлагается «All Files» (все файлы), заданный шаблоном «*.*». К нему можно добавить, скажем, «программы», задав шаблон «*.exe». Теперь остается запустить приложение и убедиться, что изменение фильтра непосредственно влияет на содержимое списка (рис. 15.2).

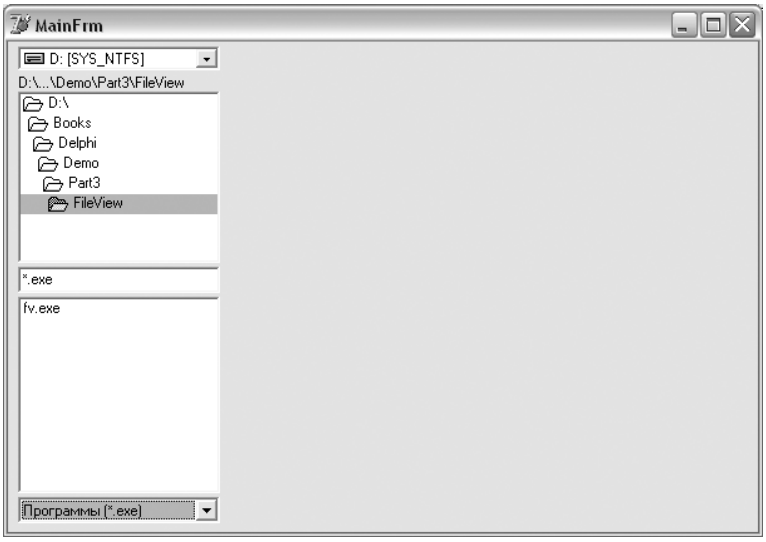

Рис. 15.2. Полноценный просмотр содержимого дисков ПК

Вновь отметим тот факт, что ни одной строчки кода для этого приложения так и не было написано!

Отображение графических изображений

Для отображения графических изображений используют компонент Image – изображение, относящийся к группе дополнительных (Additional) компонент. Обычно его помещают на поверхность формы и используют для отображения рисунков, в том числе, хранящихся в виде файлов. Компонент Image является наследником класса TGraphicControl и имеет несколько собственных свойств, связанных с графикой. Прежде всего, это самое главное свойство Image – Picture, которое служит для непосредственной работы с изображениями и имеет тип TPicture. Позже мы рассмотрим этот класс подробнее, пока же примем к сведению, что все операции над самим изображением (включая загрузку файла) выполняется посредством этого свойства.

Остальные свойства Image относятся к способам размещения изображения в рамках данного компонента. Так, свойство Center определяет, должно ли изображение быть размещено по центру (истина), или нет (ложь), в последнем случае оно будет выровнено по левому верхнему углу. Еще 2 свойства – Stretch и Proportional отвечают за масштабирование изображения. Так, если свойство Stretch установлено в истину, то изображение будет занимать все пространство, отведенное для компонента Image. Если оно при этом меньше, то оно будет увеличено, если больше – то сжато. При этом пропорции (отношение ширины к высоте) самого изображения не учитываются, что обычно приводит к искажениям. Поэтому, начиная с Delphi 6, было введено еще одно свойство – Proportional. Если оно установлено в истину, то изображение в любом случае сохранит свои исходные пропорции. Кроме того, если оно меньше, чем область, отведенная для изображения, то оно останется в исходных размерах, если только свойство Stretch так же не установлено в истину. Если же рисунок больше, то, вне зависимости от значения, заданного для Stretch, он будет пропорционально уменьшен до такого размера, чтобы поместиться в области компонента Image.

Свойство Transparent отвечает за прозрачность фоновых участков изображения: когда оно установлено в истину, фоновый цвет на рисунке заменяется прозрачным фоном. Правда, данный эффект распространяется не на все типы изображений.

Наконец, еще одно свойство, IncrementalDisplay, позволяет выбирать режим показа изображений: если установить его в истину, то файлы будут отображаться постепенно, по мере считывания и раскодирования. Это может быть актуально для больших сжатых файлов, например, для 5-мегапиксельных JPEG-изображений, особенно сохраненных в высоком качестве. При этом можно получать информацию о ходе загрузки изображения при помощи обработки события OnProgress. Это единственное собственное событие рассматриваемого компонента. Оно имеет тип TProgressEvent, для которого определен следующий синтаксис:

```
procedure (Sender: TObject; Stage: TProgressStage; PercentDone: Byte;
RedrawNow: Boolean; const R: TRect; const Msg: string) of object;
```

Аргумент Sender, как обычно, ссылается на объект, получивший уведомление о событии. Приблизительный процент выполнения операции можно получить, обратившись к PersentDone, а общее состояние – начало загрузки (psStarting), выполнение загрузки (psRunning) и завершение загрузки (psEnding) – к Stage. Ну а RedrawNow показывает, может ли быть отображена уже загруженная часть изображения, определяемая координатами прямоугольника R. На практике обычно используют только PercentDone, да и то следует учитывать, что это актуально только для файлов JPEG:

```
ProgressBar1.Position:=PercentDone;
```

Создадим простейшее приложение, которое может отображать рисунки. Для этого на форме, помимо Image, нам понадобятся следующие компоненты:

- Рамка (Bevel), чтобы визуально ограничить отведенную для рисунка область;
- Диалог открытия файла и кнопка для его вызова. Для диалога в качестве фильтра можно задать «Изображения |*.bmp;*.ico;*.wmf;*.emf;*.jpeg;*.jpg»;
- Индикатор выполнения, чтобы отображать ход загрузки JPEG-файлов.

По той причине, что стандартными средствами изображения типа JPEG не поддерживаются, то для того, чтобы программа могла с ними работать, в список используемых модулей следует добавить jpeg.

Сам код программы сводится всего лишь к 2 процедурам: обработчику OnClock копки, и обработчику OnProgress изображения. В результате мы получим код, приведенный в листинге 15.1.

Листинг 15.1. Простейшее приложение для просмотра рисунков

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, ComCtrls, StdCtrls, jpeg;
type
  TForm1 = class(TForm)
    Img: TImage;
    OpenBtn: TButton;
    ProgressBar: TProgressBar;
    OpenDlg: TOpenDialog;
    Bevel1: TBevel;
    procedure OpenBtnClick(Sender: TObject);
    procedure ImgProgress(Sender: TObject; Stage: TProgressStage;
PercentDone: Byte; RedrawNow: Boolean; const R: TRect; const Msg: String);
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
procedure TForm1.OpenBtnClick(Sender: TObject);
begin
 if OpenDlg.Execute then Img.Picture.LoadFromFile(OpenDlg.FileName);
end;
procedure TForm1.ImgProgress(Sender: TObject; Stage: TProgressStage;
PercentDone: Byte; RedrawNow: Boolean; const R: TRect; const Msg: String);
begin
 ProgressBar.Position:=PercentDone;
 if RedrawNow then Img.Repaint;
end;
end.
```

Чтобы можно было просматривать достаточно большие рисунки, не помещающиеся целиком в отведенную область, следует установить свойство Proportional компонента Img в истину, а для лучшего эстетического восприятия не помешает его еще и разместить по центру, установив в истину так же и свойство Center. Этот пример можно найти в каталоге Demo\Part3\SimpleImg.

Классы TPicture и TGraphic

Как уже было отмечено, важнейшей частью компонента Image является свойство Picture (рисунок), имеющее тип TPicture. Именно оно используется для работы с самим изображением, включая такие операции, как чтение из файла и запись в файл. Оригинальный размер изображения так же доступен через свойства Height и Width

рисунка. В свою очередь, класс TPicture включает в себя несколько свойств, основанных на классе TGraphic, которые собственно и могут содержать само изображение. Всего предусмотрено несколько таких свойств:

- Bitmap типа TBitmap – для файлов типа BMP;
- Icon типа TIcon – для файлов типа ICO;
- MetaFile типа TMetaFile – для файлов типа WMF или EMF.

Кроме них, существует еще свойство Graphic, которое ссылается на картинку вне зависимости от конкретного формата и имеет такие общие для всех рисунков свойства, как Height, Width и Transparent. Поэтому в тех случаях, когда тип изображения заведомо неизвестен, следует использовать именно это свойство. Более того, если используются нестандартные для Windows и Delphi типы графических фалов (взять тот же JPEG, который является опциональным в VCL), то только через свойство Graphic к нему и можно будет обращаться для того, чтобы, скажем, узнать размеры. Собственно говоря, когда используются свойства Height или Width объекта Picture, то они берут значения как раз от этого объекта.

Вообще можно отметить, что компонент Image является всего лишь полем для отображения изображений, поддержка которых полностью возложена на класс TPicture и его дочерние свойства, основанные на TGraphic. Единственной его особенностью являются сервисные функции для масштабирования изображений. В то же время, если этого не требуется, то можно просто создать программными методами экземпляр класса TPicture и выводить изображение на поверхность любого компонента, имеющего свойство Canvas:

```
var Pic: TPicture;
...
Pic:=TPicture.Create;
Pic.LoadFromFile('c:\mypicture.bmp');
Form1.Canvas.Draw(1,1,Pic.Graphic);
```

Кроме того, «сам по себе» класс TPicture может быть полезным при работе с таким компонентом, как ImageList. Например, можно сделать так, чтобы рисунки для изображений меню или панелей инструментов загружались из отдельного файла. В таком случае даже помещать ImageList на форму во время проектирования приложения, в обще-то не обязательно – все можно делать во время работы приложения. Кроме того, поскольку формат файл заранее известен – это будет BMP, то использовать класс TPicture так же нет необходимости, проще будет сразу обратиться к классу TBitmap – все равно TPicture будет просто переадресовывать все обращения к своему свойству Bitmap:

```
var
 ImgLst: TImageList;
 Bmp: TBitmap;
...
ImgLst:=TImageList.Create;
Bmp:=TPicture.Create;
Bmp.LoadFromFile('tools.bmp');
Bmp.Transparent:=true;
ImgList.AddMasked(Bmp,Bmp.TransparentColor);
```

Но в ряде иных случаев использование самого класса TPicture может быть более предпочтительным по той причине, что таким образом можно обеспечить поддержку других форматов. Для примера рассмотрим приложение, которое сможет не только показывать, но и создавать графические файлы. Для этого нам понадобится создать новое приложение, на главной форме которого расположим какой-либо объект, на поверхности которого можно рисовать, скажем, тот же Image. По традиции, назовем форму MainFrm, а изображение – Img. Затем поместим на форме 2 кнопки и назовем одну InsertBtn (вставить), а другую – SaveBtn (сохранить). Так же нам потребуется 2 диалога – для того, чтобы открывать и сохранять картинки. Используем специализированные диалоги OpenPictureDialog (OpenPicDlg) и SavePictureDialog (SavePicDlg).

Для того чтобы область, выделенная для творчества, была изначально заполнена белым фоном, имитируя лист бумаги, для события OnCreate или OnShow формы можно предусмотреть следующий код:

```
Img.Canvas.Brush.Color:=clWhite;
Img.Canvas.FillRect(Rect(0,0,Img.Width,Img.Height));
```

После проделанной подготовительно работы напишем код для кнопки вставки, которая будет вставлять в центр рисунка изображение из выбранного файла:

```
procedure TMainFrm.InsertBtnClick(Sender: TObject);
var
 Pic: TPicture;
begin
 if not OpenPicDlg.Execute then exit;
 Pic:=TPicture.Create;
 Pic.LoadFromFile(OpenPicDlg.FileName);
 Img.Canvas.Draw((Img.Width-Pic.Width) div 2,(Img.Height-Pic.Height) div
2,Pic.Graphic);
 Pic.Free;
end;
```

Таким образом, при помощи метода Draw холста, мы можем вывести любое количество рисунков из файлов разных типов. В данном случае все они будут центрированы по середине изображения, но при необходимости можно было бы выводить их, скажем, в то место, где был произведен последний щелчок мышью.

Но это еще не все: все-таки составлять коллажи – не единственная возможность, предоставляемая холстом. Можно на нем просто рисовать, что уже было рассмотрено в предыдущей части книги. Между тем, мы можем не просто выводить какие-либо предопределенные фигуры, но и предоставить пользователю возможность нарисовать что-либо. Пусть это будут хотя бы простые линии, которые будут появляться после того, как пользователь нажмет на левую клавишу мышки, переместит указатель, и отпустит клавишу. Для этого мы задействуем обработчики событий OnMouseDown и OnMouseUp для объекта Img, и используем методы MoveTo и LineTo его холста:

```
procedure TMainFrm.ImgMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 Img.Canvas.MoveTo(X,Y);
end;
procedure TMainFrm.ImgMouseUp(Sender: TObject; Button: TMouseButton;
```

```
  Shift: TShiftState; X, Y: Integer);
begin
 Img.Canvas.LineTo(X,Y);
end;
```

Наконец, нам остается реализовать возможность сохранения того, что в данный момент отображается в Img. Для этого нам и пригодится 2-я кнопка, код для которой будет выглядеть следующим образом:

```
if SavePicDlg.Execute then Img.Picture.SaveToFile(SavePicDlg.FileName);
```

Таким образом, все, что нарисовано на холсте, может быть без каких-либо ухищрений сохранено в качестве обычного файла в формате Windows Bitmap. Исходный код этой программы можно найти в каталоге Demo\Part3\PicEdit.

Пример программы просмотра графики

Теперь, когда мы разобрались с графикой в Delphi, попробуем создать более удобную программу для просмотра графических изображений. За основу возьмем программу для просмотра файлов – FileView, поскольку она уже содержит все необходимое для того, чтобы пользователь мог осуществлять навигацию по файловой системе своего ПК. Но прежде нам понадобится сделать некоторые изменения, в частности, было бы полезно переместить все имеющиеся компоненты непосредственно с поверхности формы на панель. Для этого сначала выделим все компоненты формы и вырежем их в буфер обмена (Ctrl+X). Затем поместим на форму панель и для ее свойства Align установим значение alLeft, в результате чего она займет все пространство по левому краю формы. Убедившись, что панель в данный момент является выбранным компонентом, произведем вставку из буфера обмена (Ctrl+V). Все ранее удаленные с поверхности формы компоненты появятся в том же виде на поверхности панели.

Разобравшись с компонентами, обеспечивающими навигацию по файлам, на пустующую правую часть формы поместим компонент ScrollBox, который находится на закладке Additional палитры компонентов. Этот компонент позволяет размещать объекты, размеры которых могут превышать отведенное для них на форме пространство. Чтобы он занял все оставшееся место, установим для него значение свойства Align в alClient, после чего поместим на него компонент Image, выровняв его по левому верхнему углу (т.е. установив свойства Left и Top в 0). Чтобы компонент мог отобразить изображение любого размера, установим его свойство AutoSize в истину.

После этого нам остается внести некоторые изменения в свойство Filter компонента-списка фильтров (назовем его FilterCb). А именно, при помощи редактора фильтра, вместо первой строки «Все программы», напишем «Все изображения» и установим следующий список расширений: «*.jpg;*.jpeg;*.bmp;*.ico;*.emf;*.wmf», после чего не забудем дописать модуль jpeg в uses. Так же можно предусмотреть возможность отбора всех этих типы файлов по отдельности, например, «Изображения в формате JPEG» с фильтром «*.jpg;*.jpeg».

Убедимся, что все компоненты правильно именованы (скажем, Image – Img, a ScrollBox – ImgSb) и приступим к написанию кода – на сей раз нам без этого не обойтись, хотя кода и потребуется всего одна строчка:

```
Img.Picture.LoadFromFile(FileLst.FileName);
```

Вопрос может быть только в том, в ответ на какое событие должен исполняться данный код. Вариантов может быт несколько, но ограничимся двойным щелчком мышкой по списку файлов, т.е. помести его в процедуру TMainFrm.FileLstDblClick. Таким образом, запустив приложение, можно выбрать любой графический файл из поддерживаемых программой форматов. Впрочем, учитывая настройки фильтра, только такие и будут отображаться в списке. Наконец, остается дважды щелкнуть по выбранному файлу, чтобы открыть его для просмотра. Если при этом изображение не будет помещаться целиком в области ScrollBox, то автоматически появятся полосы прокрутки (рис. 15.3).

Рис. 15.3. Программа просмотра изображений

Вместе с тем, в такой программе не помешало бы иметь возможность увидеть все изображение сразу, пусть и в уменьшенном масштабе. Реализовать переключение между режимами просмотра можно так же по двойному щелчку мышкой, но только уже не по списку файлов, а по самому изображению. Так что для элемента Img создадим следующий обработчик события OnDblClick:

```
procedure TMainFrm.ImgDblClick(Sender: TObject);
begin
 Img.Proportional:=not Img.Proportional;
 if Img.Proportional then Img.Align:=alClient else Img.Align:=alNone;
end;
```

В приведенном коде сначала значение свойства Proportional меняется на противоположное, после чего, в зависимости от того, включено ли масштабирование, изменяется свойство Align изображения. Суть тут в том, что если свойство Align компонента, помещенного в область прокрутки, установлено в alClient, то размеры этого компонента не могут быть больше, чем видимая часть области прокрутки. Если же размещение не задано (alNone), то, учитывая заданное нами изначально значение истины для свойства AutoSize элемента Img, этот компонент займет столько места, сколько необходимо для того, чтобы полностью показать все изображение.

В заключение остается добавить, что получившийся исходный код расположен в каталоге Demo\Part3\PicView.

Компонент MediaPlayer

Если для просмотра графических изображений используется обычный компонент Delphi, реализованный в самой VCL, то для просмотра или прослушивания мультимедийных данных используется системный компонент – MediaPlayer, расположенный на закладке System палитры компонентов. Суть системных компонент состоит в том, что практически все задачи по их работоспособности возлагаются на операционную систему, а в VCL лишь реализован простой интерфейс для доступа к функциям, предоставляемым ОС. В то же время, сам компонент MediaPlayer является обычным визуальным компонентом, основанным на TWinControl, а его «системность» проявляется лишь в виде функций, которые через его посредство реализуются.

Всего у этого компонента довольно много собственных свойств, что объясняется тем, что он, с одной стороны, предоставляет пользовательский интерфейс, а с другой – должен обеспечивать управление различными мультимедийными устройствами. Фактически, мы имеем дело с визуальным компонентом, обеспечивающим интерфейс и, одновременно, – с неким системным плеером. Список ряда свойств компонента MediaPlayer приведен в таблице15.2.

Таблица 15.2. Свойства компонента MediaPlayer

Свойство	Тип	Описание
AutoEnable	Boolean	Определяет, должен ли компонент автоматически делать доступными или недоступными свои кнопки
AutoOpen	Boolean	Определяет, должен ли плеер открываться автоматически при запуске приложения
AutoRewind	Boolean	Определяет, должен ли плеер переходить к началу записи перед следующим воспроизведением
ColoredButtons	TButtonSet	Определяет, какие из кнопок компонента должны быть цветными (по умолчанию все)
DeviceType	TMPDeviceTypes	Определяет тип мультимедийного устройства
Display	TWinControl	Определяет оконный элемент интерфейса для вывода изображения
DisplayRect	TRect	Определяет прямоугольную область для вывода изображения
EnabledButtons	TButtonSet	Определяет, какие из кнопок компонента должны быть доступными
Error	Longint	Указывает на код ошибки интерфайса MCI
ErrorMessage	String	Описывает тип ошибки MCI
FileName	String	Определяет имя файла для воспроизведения
Length	Longint	Указывает на продолжительность текущей записи
Mode	TMPModes	Указывает на текущий режим плеера
Position	Longint	Определяет текущую позицию воспроизведения в записи
Shareable	Boolean	Определяет, может ли выбранное устройство одно-

		временно использоваться другими приложениями
TrackLength	Longint	Указывает на продолжительность дорожки
TrackPosition	Longint	Указывает на насчальную позицию дророжки
Tracks	Longint	Указывает на число дорожек
VisibleButtons	TButtonSet	Определяет, какие из кнопок компонента должны отображаться

Начнем со свойств, относящихся к внешнему виду компонента. Прежде всего, это ColoredButtons, EnabledButtons и VisibleButtons. С помощью этих свойств можно управлять доступностью и видом отдельных кнопок компонента. Всего кнопок 9, по числу основных операций, необходимых для устройства воспроизведения типа проигрывателя CD. Если же проигрывать компакт-диски не планируется, то вполне можно обойтись без кнопки выброса (btEject), равно как может быть излишним и наличие кнопок перехода между дорожками (btNext и btPrev). Кнопка записи (btRecord) также является ненужной, если речь идет именно о плеере.

Некоторые свойства необходимы только для определенных устройств воспроизведения, например, свойство Display актуально только для данных, имеющих видеоряд. Тип устройства задается свойством DeviceType, которое может принимать одно из следующих значений:

- dtAutoSelect – тип устройства распознается автоматически;
- dtAVIVideo – клип видео в формате AVI;
- dtCDAudio – аудио CD;
- dtSequencer – файл MIDI;
- dtVideodisc – видео CD;
- dtWaveAudio – файл WAV.

В то же время, в типичном случае оставляют принятое по умолчанию значение dtAuto, поскольку, за исключением CD и MIDI, остальные типы из реально встречающихся сейчас файлов не используются (тип dtAVIvideo подразумевает именно формат AVI, а не MPEG). Поддержка конкретных типов воспроизводимых файлов полностью лежит на ОС и установленных программах-кодеках, так что компонент MediaPlayer может воспроизводить все те же данные, что и Windows Media Player.

В простейшем случае, когда требуется только возможность проигрывания Audio-CD, достаточно просто поместить компонент на форму и установить свойство DeviceType в dtCDAudio, а AutoOpen – в истину. После этого останется лишь запустить приложение и нажать на кнопку Play, и если в приводе окажется диск формата CD-DA, то начнется его воспроизведение.

Если же надо создать универсальный проигрыватель, то потребуется оставить эти свойства как принято по умолчанию – DeviceType в dtAutoSelect, а AutoOpen – в ложь. При этом надо будет предусмотреть код, который будет устанавливать значение свойства FileName, если только не стоит задача воспроизводить один и тот же файл. Более того, для обычного проигрывателя многие кнопки будут лишними. кроме того, у компонента MediaPlayer имеется ряд методов, дублирующих нажатие на кнопки. А именно, Play, Stop, Back, Next и т.д. Но наиболее важными являются мето-

ды Open и Close. Первый делает плеер доступным, открывая интерфейс MDI, а второй, наоборот, закрывает. При этом следует учитывать, что открывать интерфейс можно только после того, как определен файл. Соответственно, если допустить, что на форме имеется проигрыватель (MediaMP), диалог открытия файла (OpenDlg) и кнопка «Открыть файл», то для нее код получится следующим:

```
if not OpenDlg.Execute then exit;
MediaMP.Close;
MediaMP.FileName:=OpenDlg.FileName;
MediaMP.Open;
```

Если добавить еще строку с обращением к методу Play, то воспроизведение начнется автоматически. В то же время, если уж и создавать проигрыватель, то понадобится дать возможность остановить и продолжить просмотр. Итого получим 3 кнопки. Что касается компонента MediaPlayer, то нам от него понадобится лишь взаимодействие с системой, таким образом можно будет сделать его невидимым, установив свойство Visible в ложь. Вместе с тем, для вывода видео нам понадобится какой-либо компонент, происходящий от TWinControl, и обычная панель для этих целей вполне пригодна. Единственный вопрос может заключаться в том, что размеры видео могут быть больше, чем отведено пространства на панели. В таком случае нам поможет свойство DisplayRect, при помощи которого можно не только узнать размеры кадра, но и установить нужное. Поскольку при этом важно не нарушить пропорции, то проверять придется отдельно высоту и ширину, и вычислять нужный коэффициент масштабирования. Таким образом, нам надо будет сначала вычислить оба коэффициента, а затем в качестве окончательного взять больший.

```
var kw, kh, k: real;
...
kw:=MediaMP.DisplayRect.Right/VideoPan.Width;
kh:=MediaMP.DisplayRect.Bottom/VideoPan.Height;
if (kw>1) or (kh>1) then begin
  if kh>kw then k:=kh else k:=kw;
end else k:=1;
```

После этого останется лишь вычислить окончательные размеры изображения и установить получившиеся значения свойству DisplayRect, чтобы любой клип «вписался» в область просмотра, а новые значения можно вывести в окне заголовка (рис. 15.4).

Рис. 15.4. Воспроизведение видео MPEG4 с масштабированием

Полный код приложения находится в каталоге Demo\Part3\Mplay.

Компоненты Timer и PaintBox

По соседству с MediaPlayer на закладке System располагаются еще несколько компонентов, среди которых – Timer и PaintBox. Компонент PaintBox, или область рисования, являются упрощенным вариантом компонента Image: на нем можно рисовать, но он не содержит свойств, при помощи которых можно было бы ассоциировать с ним изображение. Фактически, основным предназначением этого компонента является вывод графики через его холст (свойство Canvas). Собственно говоря, никаких иных свойств, кроме унаследованных от класса TGraphicControl, этот компонент и не имеет, так что можно считать его практическим воплощением этого класса.

Другой компонент – Timer, или таймер – это невизуальный компонент, который инкапсулирует в себе функции Windows API по взаимодействию с системным таймером. Он имеет всего 2 свойства – Enabled и Interval. Свойство Enabled делает таймер активным или неактивным, а свойство Interval задает промежуток времени (в миллисекундах), через который будет возникать событие OnTimer. Событие OnTimer – это единственное событие этого компонента, оно происходит, через промежуток времени, заданный свойством Interval, если свойство Enabled установлено в истину.

Применение таймера может быть самым разнообразным, но первое, что приходит на ум – это реализация часов. Некоторые старожилы, возможно, помнят те времена, когда на компьютерах устанавливалась система Windows 3.1. Нас она в данном случае интересует с той точки зрения, что в ней была программа, представляющая собой часы, выглядящие как аналоговые. Попробуем реализовать такую программу для современных версий Windows, воспользовавшись компонентами Timer и PaintBox.

Для начала создадим новое приложение, назовем его Clock, а главную форму, по традиции – MainFrm. Затем поместим на форму метку, чтобы видеть время в цифровом формате (назовем ее DigitalLbl), и область рисования для отображения времени в аналоговом формате (AnalogPB). Зададим для метки крупный шрифт, а для области рисования установим размер 110 на 110 точек. Ну и, разумеется, нужен еще и сам таймер – назовем его ClockTmr.

Начнем с самого простого – вывода времени в цифровом формате. Для этого в обработчике события OnTimer достаточно написать всего одну строчку:

```
DigitalLbl.Caption:=TimeToStr(now);
```

Если учитывать, что по умолчанию принят интервал в 1 секунду, а сам таймер активен, то мы получим работающие часы. Кроме того, можно добавить вывод времени непосредственно в заголовок приложения, чтобы даже в свернутом виде можно было узнать время, взглянув на панель задач. Для этого достаточно изменять значение свойства Title глобального объекта Application:

```
Application.Title:=DigitalLbl.Caption;
```

Но, разумеется, гораздо больше кода потребуется написать для того, чтобы выводить время в аналоговом формате. Создадим для этих целей отдельную функцию–метод формы, которая будет принимать значение времени и выводить его на циферблате. Соответственно, она будет находиться в части public класса TMainFrm и будет определяться следующим образом:

```
procedure DrawClock(t: TTime);
```

Теперь займемся этой процедурой вплотную. Прежде всего, нам понадобится выводить циферблат часов. В принципе, можно его нарисовать в графическом редакторе и выводить при помощи метода Draw. Тем не менее, мы пойдем другим путем и будем его чертить программными методами. Поскольку мы имеем дело с окружностью, то для начала выведем эллипс, предварительно выбрав цвет фона:

```
AnalogPB.Canvas.Brush.Color:=clWhite;
AnalogPB.Canvas.Ellipse(3,3,108,108);
```

Затем начертим метки, но для начала определим интервал шага по окружности. Раз мы делаем часы, то пусть этот шаг будет равен 60. Соответственно, по известной формуле, мы получим размер шага (назовем его step, это должна быть переменная типа Double), равный 2π/60:

```
step:=2*pi/60;
```

Теперь можно вывести сами метки, начертим их в виде линий с интервалом в 5 минут или, если придерживаться более традиционной, часовой терминологии – по одной на каждый час. Для этого создадим цикл, который 12 раз будет чертить линию от центра окружности к ее периметру:

```
for i:=0 to 11 do begin
  AnalogPB.Canvas.MoveTo(55,55);
  x:=55+Round(52*sin(i*step*5)); //55 - центр окружности, 52 - длина линии
  y:=55-Round(52*cos(i*step*5));
  AnalogPB.Canvas.LineTo(x,y);
end;
```

В результате мы получим «пирог», поделенный на 12 частей (рис. 15.5, слева). Скроем часть линий, нарисовав поверх них еще одну окружность меньшего диаметра, чтобы получить более похожее на настоящее часы оформление (рис. 15.5, справа):

```
AnalogPB.Canvas.Ellipse(7,7,104,104);
```

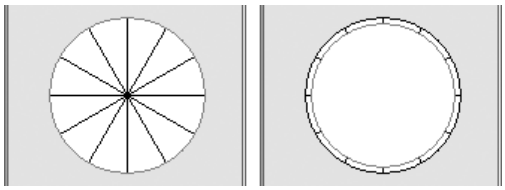

Рис. 15.5. Подготовка циферблата часов

Итак, циферблат готов, можно приступить к стрелкам. В качестве подготовительной работы разложим полученное процедурой время (t) на составляющие – часы, минуты и секунды, для чего нам понадобятся 4 целых переменных типа Word:

```
var h,m,s,ms: word;
...
DecodeTime(t,h,m,s,ms);
```

Теперь нарисуем самое простое – секундную стрелку: она должна перемещаться безо всяких дополнительных коэффициентов, строго по вычисленному шагу (step). При этом она еще и должна быть самой длинной и самой тонкой. Толщину можно уменьшить визуально при помощи более светлого, по сравнению с черным цвета, например, темно-серого. В результате мы получим следующий код:

```
AnalogPB.Canvas.MoveTo(55,55);
x:=55+Round(49*sin(s*step));
y:=55-Round(49*cos(s*step));
AnalogPB.Canvas.Pen.Color:=clGray;
AnalogPB.Canvas.LineTo(x,y);
```

Следующей будет минутная стрелка. Поскольку минут тоже 60, то, в простейшем случае, для вычисления координат окончания линии мы могли бы воспользоваться той же формулой, что и для секунд, заменив лишь переменную s на m. Но если для секундной стрелки дискретное движение нормально, то для минутной не совсем годится (а для часовой будет вообще не приемлемо). Поэтому нам придется вычислять промежуточное значение, чтобы учитывать секунды, прошедшие с начала минуты:

```
c:=(m*60+s)/60;
```

Соответственно, после этого останется вычислить координаты на основе получившегося значения, не забыв вернуть черный цвет и сделать стрелку короче:

```
x:=55+Round(43*sin(c*step));
y:=55-Round(43*cos(c*step));
AnalogPB.Canvas.Pen.Color:=clBlack;
AnalogPB.Canvas.LineTo(x,y);
```

На последок определимся с часами. Здесь, помимо уже известного решения проблемы с плавным перемещением стрелки, имеется еще 2 вопроса, а именно – необходимость повышающего коэффициента перемещения (т.к. часов меньше 60), и перевод 24-часового формата в 12-часовой. Вопрос с форматом решается просто: достаточно проверять переменную h на то, не больше ли она 12, и если да – то уменьшать ее на 12. На этом основании можно будет вводить коэффициент, равный 5 (60/12):

```
if h>12 then h:=h-12;
c:=(h*60+m)/60*5;
```

После этого остается вычислить значения и нарисовать короткую толстую стрелку, не забыв после этого вернуть толщину линии обратно:

```
x:=55+Round(33*sin(c*step));
y:=55-Round(33*cos(c*step));
AnalogPB.Canvas.Pen.Width:=2;
AnalogPB.Canvas.LineTo(x,y);
AnalogPB.Canvas.Pen.Width:=1;
```

Последним штрихом может быть изменение оформление окна – для такого приложения лучше всего будет установить свойство BorderStyle в bsToolWindow (рис. 15.6).

Рис. 15.6. Работающие «аналоговые» часы

Еще одним дополнением может стать размещение окна с часами в нижнем левом углу экрана, для чего в обработчике OnCreate главной формы следует предусмотреть соответствующий код. Итоговый код программы приведен в листинге 15.2.

Листинг 15.2. Часы

```
unit main;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls;
type
  TMainFrm = class(TForm)
    ClockTmr: TTimer;
    DigitalLbl: TLabel;
    AnalogPB: TPaintBox;
    AlphaTmr: TTimer;
    procedure ClockTmrTimer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  public
    procedure DrawClock(t: TTime);
  end;
var
  MainFrm: TMainFrm;
implementation
{$R *.dfm}
procedure TMainFrm.ClockTmrTimer(Sender: TObject);
begin
 DigitalLbl.Caption:=TimeToStr(now);
 Application.Title:=DigitalLbl.Caption;
 DrawClock(now);
end;
procedure TMainFrm.DrawClock(t: TTime);
var
 x,y,i: integer;
 h,m,s,ms: word;
 step,c: double;
begin
 AnalogPB.Canvas.Brush.Color:=clWhite;
 AnalogPB.Canvas.Ellipse(3,3,108,108);
 step:=2*pi/60;
 AnalogPB.Canvas.Pen.Color:=clBlack;
 for i:=0 to 11 do begin
   AnalogPB.Canvas.MoveTo(55,55);
   x:=55+Round(52*sin(i*step*5));
   y:=55-Round(52*cos(i*step*5));
   AnalogPB.Canvas.LineTo(x,y);
 end;
```

```
 AnalogPB.Canvas.Pen.Color:=clGray;
 AnalogPB.Canvas.Ellipse(7,7,104,104);
 DecodeTime(t,h,m,s,ms);
 AnalogPB.Canvas.MoveTo(55,55);
 x:=55+Round(49*sin(s*step));
 y:=55-Round(49*cos(s*step));
 AnalogPB.Canvas.Pen.Color:=clGray;
 AnalogPB.Canvas.LineTo(x,y);
 AnalogPB.Canvas.MoveTo(55,55);
 c:=(m*60+s)/60;
 x:=55+Round(43*sin(c*step));
 y:=55-Round(43*cos(c*step));
 AnalogPB.Canvas.Pen.Color:=clBlack;
 AnalogPB.Canvas.LineTo(x,y);
 AnalogPB.Canvas.MoveTo(55,55);
 if h>12 then h:=h-12;
 c:=(h*60+m)/60*5;
 x:=55+Round(33*sin(c*step));
 y:=55-Round(33*cos(c*step));
 AnalogPB.Canvas.Pen.Width:=2;
 AnalogPB.Canvas.LineTo(x,y);
 AnalogPB.Canvas.Pen.Width:=1;
end;
procedure TMainFrm.FormCreate(Sender: TObject);
begin
 MainFrm.Top:=Screen.DesktopHeight-MainFrm.Height;
 MainFrm.Left:=Screen.DesktopWidth-MainFrm.Width;
end;
```

Если же вам не понравится, что они таким образом закроют «стандартные» часы Windows, да еще и вместе с индикатором клавиатуры, то можно будет сделать их полупрозрачными, причем эффект появления прозрачности может быть постепенным, для чего достаточно будет разместить на форме еще один таймер и изменять по нему свойство AlphaBlendValue. Именно такой вариант программы вы найдете в Demo\Part3\Time.

Современные файловые компоненты

Начиная с Delphi 6, в состав VCL было включено несколько дополнительных компонент, обеспечивающих доступ к файловой системе. Это ShellTreeView, ShellListView и ShellComboBox, все они находятся на закладке примеров (Samples). По этой причине их принято считать нестандартными компонентами VCL, и, более того, они не упоминаются в справочной системе. Тем не менее, эти компоненты являются удобной альтернативой морально устаревшим и не отвечающим современным требованиям компонентам DirectoryListBox, FileListBox и DriveComboBox. В частности, с их помощью затруднительно добраться до таких мест системы, как рабочий стол или до папки «Мои документы». Кроме того, они не могут напрямую работать с сетью. В современных компонентах, интегрированных с ОС, эти проблемы решены. В частности, компонент ShellTreeView (системное дерево) отображает структуру папок и уст-

ройств компьютера, а так же сетевых ресурсов. Подобную задачу выполняет и ShellComboBox, с той лишь разницей, что он не отображает каталоги на дисках и не показывает иерархии объектов. А компонент ShellListView может отображать все объекты, но не их иерархию. Сами системные компоненты, реализованные в ОС, можно видеть в проводнике Windows. В нем область «папки» соответствует компоненту ShellTreeView, основная область – ShellListView, а адресная строка, с некоторыми оговорками – ShellComboBox.

Рассмотрим некоторые свойства этих компонент, и начнем с ShellTreeView, для чего обратимся к таблице 15.3.

Таблица 15.3. Свойства ShellTreeView

Свойство	Тип	Описание
AutoContextMenu	Boolean	Определяет, должно ли использоваться контекстное меню, предоставляемое ОС
AutoRefresh	Boolean	Определяет, должно ли содержимое обновляться автоматически
ObjectTypes	TShellObjectTypes	Определяет, объекты каких типов должны отображаться
Root	String	Определяет корневой каталог или системную папку
ShellComboBox	TShellComboBox	Указывает на ассоциированный компонент
ShellListView	TShellListView	Указывает на ассоциированный компонент
ShowButtons	Boolean	Определяет, отображать или нет значки рядом с именем папки, чтобы раскрыть ее содержимое
ShowLines	Boolean	Определяет, должны или нет выводиться линии
ShowRoot	Boolean	Определяет, должна или нет отображаться корневая папка

Свойство ObjectTypes имеет 3 флага, указывающих на то, какие компоненты следует показывать. Это otFolders – папки, otNonFolders – все, кроме папок, и otHidden – скрытые объекты.

Свойство Root, помимо непосредственного пути, вроде «c:\Dirname», может содержать специальные значения, ссылающиеся на различные системные папки. В их числе можно отметить следующие:

- rfDesktop – рабочий стол;
- rfMyComputer – «мой компьютер»:
- rfNetwork – «сетевое окружение»;
- rfControlPanel – панель управления;
- rfPrinters – принтеры;
- rfFonts – шрифты;
- rfAppData – данные приложений (Documents and Settings);
- rfRecycleBin – «корзина».

Компонент ShellListView имеет все те же самые свойства (из приведенных в таблице 15.3), что и ShellTreeView, разумеется, за исключением свойства ShellListView: вместо него используется аналогичное по назначению свойство ShellTreeView. Кроме того, у него имеется еще одно свойство – AutoNavigate, которое, будучи установленным в истину, позволит пользователю открывать содержимое каталогов непосредственно из этого компонента.

Наконец, компонент ShellComboBox, основанный на раскрывающемся списке, содержит только 1 свойство из всех вышеперечисленных – Root.

Для того чтобы самостоятельно создать некоторое подобие Проводника, достаточно поместить на форму все эти компоненты и связать их между собой при помощи свойств ассоциирования. А чтобы такой проводник вел себя так же, как имеющийся в Windows, т.е. мог изменять размеры, нам понадобится еще панель, на которую мы поместим ShellComboBox, и разделитель – компонент Splitter. Он находится на закладке Additional и выполняет функцию ползунка, при помощи которого можно изменять размеры соседних компонентов. При этом для всех участвующих в этом компонентов должно быть настроено выравнивание (свойство Align). Таким образом, вначале для панели, на которой находится раскрывающийся список, следует установить свойство Align в alTop, в результате чего панель займет все пространство вверху окна. Затем для дерева (ShellTreeView) установим значение Align в alLeft, чтобы оно расположилось по левому краю окна. После этого поместим на форму разделитель. Поскольку он должен граничить как раз с деревом, и изменять, в том числе, и его размеры, для него свойство Align тоже должно быть установлено в alLeft. Впрочем, по умолчанию оно как раз такое значение и имеет. Наконец, чтобы занять все оставшееся место, установим для ShellComboBox выравнивание в alClient. Запустив программу, можно убедиться, что она действительно функционирует подобно Проводнику Windows (рис. 15.7).

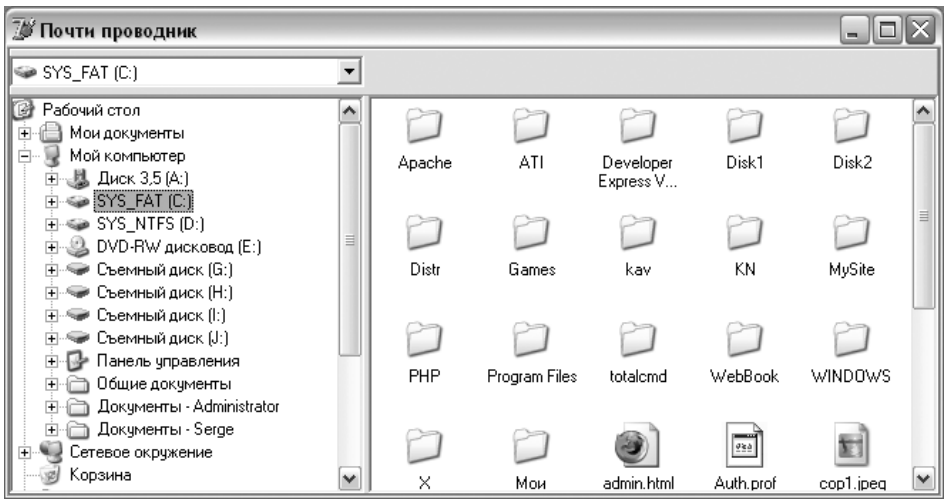

Рис. 15.7. Упрощенный аналог проводника Windows

Подобно «настоящей» области просмотра файлов, ShellListView можно переключать между режимами отображения, для чего следует воспользоваться свойством ViewStyle, которое может принимать одно из следующих значений: vsIcon, vsSmallIcon,

vsList и vsReport. В примере, находящемся в Demo\Part3\Explore, продемонстрированы все эти режимы применительно к нашему проводнику.

ПРИМЕЧАНИЕ

Следует отметить, что свойство ViewStyle происходит от предка ShellListView – универсального, с точки зрения применения, компонента ListView. Так же существует и прообраз ShellTreeView – компонент TreeView. Оба они относятся к группе Win32 и могут быть использованы для предоставления в иерархическом виде самой разнообразной информации.

Дополнительные компоненты

Помимо уже рассмотренных компонентов, VCL включает в себя ряд иных компонент, одни из которых, например, кнопка с рисунком (BitButton), являются просто расширенными версиями стандартных элементов управления, а другие представляют собой что-то принципиально новое – например, таблицы Grid. Кроме того, библиотеку VCL можно пополнять путем установки дополнительных компонент, как разработанных третьими разработчиками, так и собственными. Наконец, Delphi поддерживает технологию ActiveX, что позволяет внедрять целые приложения непосредственно в разрабатываемую программу.

Кнопки

Помимо стандартной кнопки (Button), в VCL имеются еще 2 компонента, реализующих этот важный элемент интерфейса. Один из них, компонент BitButton, или кнопка с рисунком, является расширенным вариантом кнопки. Этот компонент находится на закладке Advanced палитры инструментов.

Основное отличие этого компонента по сравнению с Button состоит в том, что помимо текстовой надписи, BitButton может содержать изображение, задаваемое через свойство Glyph типа TBitmap. Это изображение можно разместить слева от текста, справа, снизу, или сверху, за что отвечает свойство Layout (см. рис. 16.1). Кроме того, предусмотрено 10 стандартных типов этих кнопок для всех часто встречающихся операций в модальных диалоговых окнах. Тип задается при помощи свойства Kind, и не только добавляет соответствующие надпись и рисунок, но и устанавливает в необходимое значение свойство ModalResult (рис. 16.1).

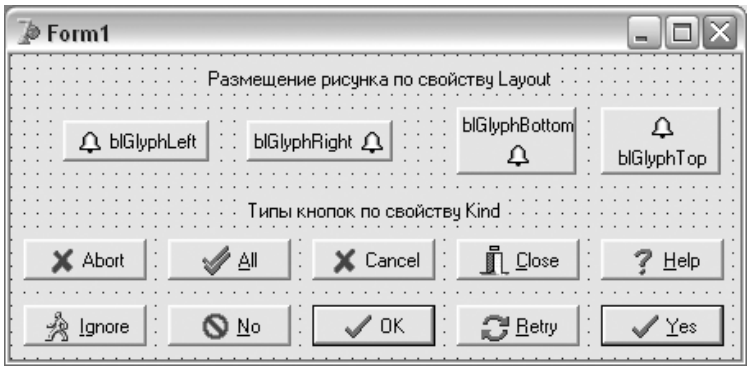

Рис. 16.1. Выравнивание рисунка и типы кнопок

Для более точного позиционирования картинки относительно надписи можно использовать свойство Spacing, а при помощи свойства Margin можно регулировать отступы от рисунка до края кнопки.

Еще одно дополнительное свойство компонента BitButton – NumGlyphs, позволяет использовать составные картинки. Например, если использовать рисунок размером 20 на 40 пикселей, то при NumGlyphs, установленном в 2, в обычных условиях будет отображаться его левая половина (фрагмент 20 на 20). Если же сделать кнопку недоступной (свойство Enabled), то отобразится вторая, правая часть рисунка. Таким образом можно визуально подчеркивать состояние кнопки.

Другой вариант кнопки, располагающийся по соседству с BitButton на палитре компонентов – это SpeedButton, или «быстрая кнопка». Обычно такой элемент интерфейса используют для вызова команд или установки какого-либо режима. В последнем случае довольно часто применяют триггерный эффект, для чего объединяют несколько кнопок в группу. Для этих целей у компонента SpeedButton предусмотрено свойство GroupIndex: все такие кнопки, имеющие одно и тот же значение этого свойства, отличное от 0, объединяются в общую группу. При этом узнать, какая кнопка в группе является выбранной, можно при помощи свойства Down: если оно установлено в истину, значит, данная кнопка выбрана («нажата»).

Другими собственными свойствами компонента SpeedButton являются уже знакомые нам по BitButton свойства Glyph, Layout, Margin, NumGlyphs и Spacing. Их предназначение полностью соответствует таковому у одноименных свойств у BitButton.

Важным отличием компонента SpeedButton от кнопки с рисунком, как, впрочем, и от обычной кнопки Button, является то, что он основан на классе TGraphicControl. Таким образом, несмотря на ряд общих с BitButton черт, этот компонент является совершенно другим по своей сути, и, прежде всего, не может получить фокус ввода с клавиатуры. Соответственно, SpeedButton следует использовать в тех случаях, когда получение фокуса ввода кнопкой нежелательно. Как правило, это кнопки на панелях инструментов.

Кроме того, у «быстрой кнопки» имеется еще одно свойство – Flat, которое определяет внешний вид такой кнопки. Если это свойство установлено в истину, то сама кнопка становится прозрачной, и визуально выделяется лишь во время выполнения программы – при наведении на нее указателя мышки.

Таблицы

Как известно, таблицы представляют собой удобное средство для представления данных, разбитых на строки и столбцы. На закладке дополнительных компонент вы найдете 2 компонента, являющихся реализацией таблиц в VCL. Это StringGrid и DrawGrid. Оба они позволяют выводить любые данные, включая текст и изображения в свои ячейки, однако компонент DrawGrid автоматически выводит лишь сетку, разделяющую ячейки, а вывод собственно информации следует программировать самостоятельно. Что касается компонента StringGrid, то с его помощью без лишних хлопот можно выводить в ячейки текстовую информацию.

Математически таблицу можно представить как 2-мерный массив (матрицу). Именно так и организовано хранение текстовой информации у компонента StringGrid, одним из наиболее важных свойств которого можно считать Cells. Это свойство, доступное

только во время выполнения, определено как матрица строк, где 1-й индекс определяет столбец, а 2-й – ряд:

```
Cells[ACol, ARow: Integer]: string;
```

Интересной особенностью компонента StringGrid является то, что помимо строки, для каждой ячейки он может хранить еще и произвольный объект. Доступ к объектам можно получить через свойство Objects, определенное аналогичным образом:

```
Objects[ACol, ARow: Integer]: TObject;
```

Для удобства доступа к информации определены еще 2 свойства – Cols и Rows, которые представляют собой одномерные массивы столбцов и рядов таблицы, представленных в виде объектов TStrings:

```
Cols[Index: Integer]: TStrings;
Rows[Index: Integer]: TStrings;
```

Таким образом, при помощи этих свойств можно получить доступ сразу как к строковому содержимому ячеек, так и к объектам.

Все эти свойства позволяют достаточно легко манипулировать содержимым таблицы, что применительно к StringGreed, однако у обоих типов таблиц имеется целый ряд общих свойств, касающихся как их внешнего представления, так и используемых для контроля их содержимого. Они приведены в таблице 16.1.

Таблица 16.1. Свойства таблиц

Свойство	Тип	Описание
BorderStyle	TBorderStyle	Определяет, должна ли быть рамка вокруг таблицы
Col	Longint	Указывает на индекс столбца, в котором находится выбранная в данный момент ячейка
ColCount	Longint	Определяет количество столбцов в таблице
ColWidths	массив Integer	Определяет ширину каждого столбца в пикселях
DefaultColWidth	Integer	Определяет ширину столбцов по умолчанию
DefaultRowHeight	Integer	Определяет высоту рядов по умолчанию
EditorMode	Boolean	Определяет режим текущей ячейки: истина - режим правки, ложь - режим просмотра
FixedColor	TColor	Определяет цвет фона для фиксированных (заголовочных) ячеек
FixedCols	Integer	Определяет число фиксированных столбцов
FixedRows	Integer	Определяет число фиксированных рядов
GridLineWidth	Integer	Определяет толщину разделяющих ячейки линий
LeftCol	Longint	Определяет первый видимый столбец таблицы слева
Options	TGridOptions	Определяют параметры вида и поведения таблицы
Row	Longint	Указывает на индекс ряда, в котором находится выбранная в данный момент ячейка
RowCount	Longint	Определяет количество рядов в таблице
RowHeights	массив Integer	Определяет высоту каждого ряда в пикселях
ScrollBars	TScrollStyle	Определяет наличие полос прокрутки: ssNone, ssHorizontal, ssVertical, ssBoth

TopRow	Longint	Определяет первый видимый ряд таблицы сверху
VisibleColCount	Integer	Указывает на количество одновременно видимых столбцов
VisibleRowCount	Integer	Указывает на количество одновременно видимых рядов

Применительно к приведенным свойствам следует сделать 2 пояснения. Прежде всего, каждая таблица может содержать заголовочные ячейки, которые не прокручиваются вместе с остальными ячейками таблицы и видны всегда. Это характерно для любых электронных таблиц: например в Excel это ячейки, содержащие номера рядов и столбцов. При этом по умолчанию определено по 1 ряду и столбцу – как в Excel, однако вы можете изменить эти значения, отказавшись от заголовков вообще или наоборот, добавив дополнительные фиксированные ряды или столбцы.

Так же отдельных комментариев требует свойство Options. Подобно одноименному свойству других компонент, оно является набором флагов, позволяющих установить ряд специфических особенностей данного компонента. Применительно к таблицам мы имеем следующий набор:

- goFixedVertLine – Вертикальные линии-разделители будут выводиться между столбцами фиксированных ячеек;
- goFixedHorzLine – Горизонтальные линии-разделители будут выводиться между рядами фиксированных ячеек;
- goVertLine – Вертикальные линии-разделители будут выводиться между столбцами обычных ячеек;
- goHorzLine – Вертикальные линии-разделители будут выводиться между рядами обычных ячеек;
- goRangeSelect – Пользователь сможет выделить диапазон ячеек сразу, если не установлен флаг goEditing;
- goDrawFocusSelected – Выбранная ячейка должна выделяться фоновой заливкой, а не только рамкой;
- goRowSizing – Позволяет изменять высоту рядов;
- goColSizing – Позволяет изменять ширину столбцов;
- goRowMoving – Позволяет перетаскивать столбцы при помощи мышки;
- goColMoving – Позволяет перетаскивать ряды при помощи мышки;
- goEditing – Делает возможной правку ячеек пользователем;
- goTabs – Позволяет перемещаться между ячейками при помощи клавиши табуляции;
- goRowSelect – При выборе ячейки будет выделяться весь ряд;
- goAlwaysShowEditor – Таблица постоянно находится в режиме правки.
- goThumbTracking – Включает режим немедленного обновления таблицы во время ее прокрутки пользователем.

В типичном случае, когда делают таблицу для построчного просмотра данных, устанавливают следующий набор опций: флаг goEditing оставляют выключенным, а goRowSelect включают. Если же требуется сделать таблицу, в которую пользователь сможет вносить изменения, то включают флаг goEditing.

Простейший табличный редактор

Чтобы лучше понять работу таблиц, рассмотрим компонент StringGrid на примере приложения, которое может редактировать таблицы, а так же сохранять их в текстовых файлах, или считывать из них. В качестве формата файла можно использовать либо текст с табуляцией, либо формат CSV (с разделителями точка с запятой). Мы реализуем поддержку обоих форматов, используя расширение файла в качестве критерия: если txt – то табуляция, а если csv – значит точка с запятой.

Для нового приложения используем 2 основных компонента – панель инструментов (ToolBar) и собственно таблицу (StringGrid). При помещении на форму компонент ToolBar – назовем его MainTb – автоматически примет выравнивание по верхнему краю. Что касается таблицы, то для нее мы установим свойство Align в alClient, чтобы она заняла все оставшееся пространство. Чтобы вид и возможности таблицы лучше соответствовали предназначению нашей программы, уменьшим высоту ряда до 20 пикселей (свойство DefaultRowHeight), а ширину колонок сделаем изменяемой, установив включенным флаг goColSizing в свойстве Options. Так же удалим фиксированную колонку, установив свойство FixedCols в 0.

Теперь вернемся к панели инструментов. Как минимум, нам понадобятся 2 кнопки: для открытия и сохранения файла. Вместе с тем, поскольку изначально отведено всего по 5 строк и рядов, то не помешает дать пользователю возможность добавлять дополнительные строки и ряды. Сделаем обе эти функции на одной кнопке, воспользовавшись ее способностью к совмещению с всплывающим меню. Для этого установим свойство Style 3-й кнопки в tbsDropDown, и разместим на форме еще один компонент, PopupMenu, назвав его, скажем, AddRowColDdm, после чего присвоим свойству DropDownMenu этой кнопки значение AddRowColDdm.

Наконец, предусмотрим еще и различные режимы работы с таблицей – только просмотр или просмотр в правкой. Для этого на панели инструментов нам понадобятся еще 2 кнопки, для которых мы реализуем триггерный эффект. Для этого у обеих кнопок следует установить свойство Grouped в истину, а свойство Style – в tbsCheck. На этом же этапе следует определиться с режимом по умолчанию. Оставим его таким, как есть, т.е. без возможности правки. Соответственно, для кнопки, включающей режим «только просмотр» установим свойство Down в истину. Еще одна кнопка, которую, при желании, можно добавить на панель инструментов – это кнопка выхода из программы.

Поскольку по общепринятому правилу кнопки на панели инструментов должны иметь графические символы, изображающие действие, которое они выполняют, то поместим на форму объект ImageList, заполним его картинками и ассоциируем с панелью инструментов путем присвоения имени списка изображений свойству Images.

Если теперь еще раз посмотреть на форму и панель инструментов, то можно отметить, что нам не хватает еще 2 компонентов, а именно – диалогов открытия и сохранения файлов. Поместим 2 соответствующих компонента на форму, завершив, таким образом, подготовительную работу по интерфейсу (рис. 16.2).

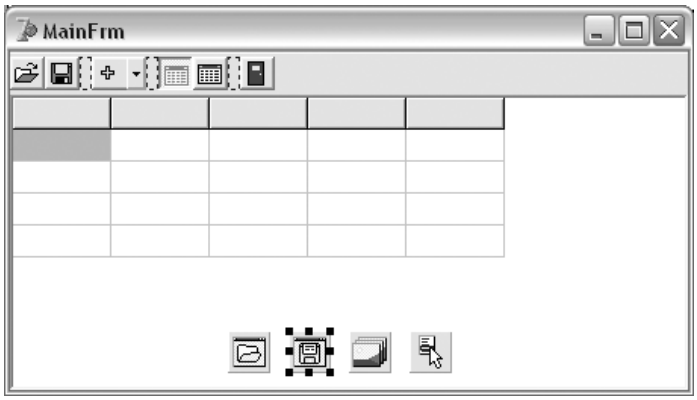

Рис. 16.2. Визуальная часть разработки табличного редактора

Следующим этапом будет разработка методов загрузки и сохранения таблиц. Начнем с загрузки, тем более что тестовую таблицу можно создать и сохранить в любом табличном редакторе, скажем, в Excel или в Calc.

Для реализации функции считывания файла и размещения информации в ячейках таблицы, воспользуемся классом TStringList и предоставляемой им функциональностью. В частности, для того чтобы считать файл, достаточно использовать его метод LoadFromFile. После этого останется лишь обработать каждую строку, представив ее так же в виде объекта класса TStringList с тем, чтобы присвоить его в качестве значения свойства Rows таблицы. Для этого нам понадобится сервисная функция, которая и будет делать данную операцию. Назовем ее StringToList:

```
function StringToList(str: string; sep: char):TStringList;
var
 x: integer;
 s: string;
begin
 Result:=TStringList.Create;
 repeat
   x:=pos(sep,str);
   if x>0 then Result.Add(copy(str,1,x-1)) else begin
     s:=copy(str,1,length(str));
     if s<>'' then Result.Add(s);
   end;
   delete(str,1,x);
 until x=0;
end;
```

Данная функция принимает 2 аргумента: саму строку для преобразования, а так же символ-разделитель, по которому строка должна будет разделяться на элементы списка. Применительно к нашей программе в качестве такого символа может использоваться либо знак табуляции (#9), либо точка с запятой (#59). В самом начале создается экземпляр класса TStringList для результата функции, после чего выполняется цикл с постусловием до тех пор, пока в строке находится хоть 1 разделитель. Эта проверка выполняется первой в теле цикла, после чего нужная часть строки копиру-

ется и вставляется в список. Таким образом, на выходе мы получаем список строк, который можно будет присвоить свойству Rows. Вместе с тем, учитывая, что строк, как правило, будет больше одной, вызов функции StringToList так же следует разместить в цикле, который будет повторяться по всем строкам того списка, в который изначально был загружен файл. В результате мы получаем примерный набросок кода самой процедуры загрузки файла:

```
procedure TMainFrm.LoadData(fn: string);
var
 lst, rlst: TStringList;
 sep: Char;
 i: integer;
begin
 lst:=TStringList.Create;
 lst.LoadFromFile(fn);
 if ExtractFileExt(fn)='.csv' then sep:=#59 else sep:=#9;
 for i:=1 to lst.Count do begin
   rlst:=StringToList(lst[i-1],sep);
   DataSg.Rows[i]:=rlst;
 end;
 lst.Free;
end;
```

Отметим, что поскольку данная процедура взаимодействует с элементами формы, то мы сделали ее методом класса MainFrm. Стандартная функция ExtractFileExt используется для определения того, какой разделитель следует применить, в зависимости от расширения файла, после чего функция StringToList вызывается с нужным значением. В завершение своей работы эта процедура освобождает занимаемую списком память. Таким образом, обработчик события OnClick для кнопки загрузки таблицы будет выглядеть таким образом:

```
if not OpenDlg.Execute then exit;
LoadData(OpenDlg.FileName);
Caption:=OpenDlg.FileName;
```

В результате после выбора файла при помощи стандартного диалога он будет загружен в таблицу, а заголовок формы будет содержать его полное имя.

Вместе с тем, данная процедура не лишена недостатков. Прежде всего, она не устанавливает размеры таблицы в соответствии с реальными размерами загруженного файла. Кроме того, следует учитывать, что в процессе считывания файла могут возникать ошибки – попробуйте открыть тестовый файл сначала в Excel, а затем, не закрывая его в нем, попытаться открыть в созданной программе, и вы получите ошибку доступа к файлу. Таким образом, нам надо, во-первых, изменять размеры таблицы, а во-вторых – заключить блок обработки в оператор try…finally:

```
lst:=TStringList.Create;
try
  lst.LoadFromFile(fn);
  if ExtractFileExt(fn)='.csv' then sep:=#59 else sep:=#9;
  for i:=1 to lst.Count do begin
```

```
    rlst:=StringToList(lst[i-1],sep);
    DataSg.Rows[i]:=rlst;
    if rlst.Count>DataSg.ColCount then DataSg.ColCount:=rlst.Count;
  end;
  DataSg.RowCount:=lst.Count+1;
finally
  lst.Free;
end;
```

Более того, для того, чтобы проинформировать вызывающую процедуру о результате выполнения этой подпрограммы, нам нужна не процедура, а функция с возвращаемым значением типа Boolean. При этом значение результата в истину следует разместить в последней строке, после блока try…finally: если произойдет ошибка, то до выполнения присваивания «Result:=true» дело не дойдет. Сама же вызывающая процедура (обработчик события OnClick) может реагировать на удачную загрузку файла выводом его пути в строке заголовка окна, а в случае неудачи – не делать ничего:

```
if LoadData(OpenDlg.FileName) then Caption:=OpenDlg.FileName;
```

Теперь рассмотрим сохранение данных. Оно должно работать с точностью до наоборот: вместо раскладывания сроки на список, она должна делать строку с разделителями из списка и сохранять результаты в указанный файл. Процедура обратного преобразования получается гораздо проще – достаточно выполнить цикл по количеству элементов списка, всякий раз добавляя в конец строки указанный разделитель:

```
function ListToString(lst: TStringList; sep: Char):string;
var
 i: integer;
begin
 Result:='';
 for i:=0 to lst.Count-1 do begin
   Result:=Result+lst[i]+sep;
 end;
end;
```

Соответственно, сама процедура сохранения должна будет последовательно обойти все ряды таблицы, «склеивая» ячейки в строку через разделитель. Сделать это, опять-таки, можно через свойство Rows. Единственным подводным камнем здесь является то, что свойство Rows[], на самом деле, представлено абстрактным классом TStrings. Впрочем, это ограничение легко обходится путем явного приведения класса TStrings к TStringList:

```
for i:=1 to DataSg.RowCount do begin
  lst.Add(ListToString(TStringList(DataSg.Rows[i]),sep));
end;
```

После этого останется лишь создать обработчики для оставшихся кнопок. В частности, для триггеров, переключающих таблицу между режимами просмотра и правки, следует написать код, который будет добавлять, или наоборот, удалять флаг goEditing из свойства Options. А для добавления столбцов или рядов достаточно увеличивать на 1 значение свойств ColCount и RowCount. Итоговый код программы приведен в листинге 16.1.

Листинг 16.1. Табличный редактор

```
unit main;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ImgList, ComCtrls, ToolWin, Grids, Menus;
type
  TMainFrm = class(TForm)
    DataSg: TStringGrid;
    MainTb: TToolBar;
    OpenBtn: TToolButton;
    SaveBtn: TToolButton;
    AddBtn: TToolButton;
    OpenDlg: TOpenDialog;
    SaveDlg: TSaveDialog;
    MainTbIl: TImageList;
    AddRowColDdm: TPopupMenu;
    AddRowM: TMenuItem;
    AddColM: TMenuItem;
    ToolButton1: TToolButton;
    ViewBtn: TToolButton;
    EditBtn: TToolButton;
    ToolButton2: TToolButton;
    CloseBtn: TToolButton;
    ToolButton3: TToolButton;
    procedure OpenBtnClick(Sender: TObject);
    procedure SaveBtnClick(Sender: TObject);
    procedure AddRowMClick(Sender: TObject);
    procedure AddColMClick(Sender: TObject);
    procedure ViewBtnClick(Sender: TObject);
    procedure EditBtnClick(Sender: TObject);
    procedure CloseBtnClick(Sender: TObject);
  public
    function LoadData(fn: string):boolean;
    function SaveData(fn: string):boolean;
  end;
var
  MainFrm: TMainFrm;
implementation
{$R *.dfm}
function StringToList(str: string; sep: char):TStringList;
var
 x: integer;
 s: string;
begin
 Result:=TStringList.Create;
 repeat
```

```
   x:=pos(sep,str);
   if x>0 then Result.Add(copy(str,1,x-1)) else begin
     s:=copy(str,1,length(str));
     if s<>'' then Result.Add(s);
   end;
   delete(str,1,x);
 until x=0;
end;
function ListToString(lst: TStringList; sep: Char):string;
var
 i: integer;
begin
 Result:='';
 for i:=0 to lst.Count-1 do begin
   Result:=Result+lst[i]+sep;
 end;
end;
function TMainFrm.LoadData(fn: string):boolean;
var
 lst, rlst: TStringList;
 sep: Char;
 i: integer;
begin
 lst:=TStringList.Create;
 try
   lst.LoadFromFile(fn);
   if ExtractFileExt(fn)='.csv' then sep:=#59 else sep:=#9;
   for i:=1 to lst.Count do begin
     rlst:=StringToList(lst[i-1],sep);
     DataSg.Rows[i]:=rlst;
     if rlst.Count>DataSg.ColCount then DataSg.ColCount:=rlst.Count;
   end;
   DataSg.RowCount:=lst.Count+1;
 finally
   lst.Free;
 end;
 Result:=true;
end;
function TMainFrm.SaveData(fn: string):boolean;
var
 lst: TStringList;
 sep: Char;
 i: integer;
begin
 lst:=TStringList.Create;
 if ExtractFileExt(fn)='.csv' then sep:=#59 else sep:=#9;
 for i:=1 to DataSg.RowCount do begin
```

```
    lst.Add(ListToString(TStringList(DataSg.Rows[i]),sep));
  end;
  try
    lst.SaveToFile(fn);
  finally
    lst.Free;
  end;
  Result:=true;
end;
procedure TMainFrm.OpenBtnClick(Sender: TObject);
begin
  if not OpenDlg.Execute then exit;
  if LoadData(OpenDlg.FileName) then Caption:=OpenDlg.FileName;
end;
procedure TMainFrm.SaveBtnClick(Sender: TObject);
begin
  if not SaveDlg.Execute then exit;
  if SaveData(SaveDlg.FileName) then Caption:=SaveDlg.FileName;
end;
procedure TMainFrm.AddRowMClick(Sender: TObject);
begin
  DataSg.RowCount:=DataSg.RowCount+1;
end;
procedure TMainFrm.AddColMClick(Sender: TObject);
begin
  DataSg.ColCount:=DataSg.ColCount+1;
end;
procedure TMainFrm.ViewBtnClick(Sender: TObject);
begin
  DataSg.Options:=DataSg.Options-[goEditing];
end;
procedure TMainFrm.EditBtnClick(Sender: TObject);
begin
  DataSg.Options:=DataSg.Options+[goEditing];
end;
procedure TMainFrm.CloseBtnClick(Sender: TObject);
begin
  close;
end;
end.
```

Исходный код так же можно найти в каталоге Demo\Part3\Grid.

Другие дополнительные компоненты

На закладке Additional можно обнаружить еще целый ряд компонентов. Например, ValueListEditor, является частным случаем таблицы, предназначенной для правки пар типа «имя-значение». Основными его свойствами можно считать Keys и Values. Первое представляет собой массив значений левого столбца («имён»), а второе позволяет

по имени найти соответствующее ему значение во втором столбце. Например, если в левом столбце будут следующие значения: цвет, размер и цена, а во втором – красный, 100 и $250, то выражение «Values['цвет']» даст результат «красный».

Расположенный рядом с ValueListEditor компонент LabeledEdit представляет собой «2 компонента в 1» – метку (Label) и однострочный редактор (Edit). В результате он обладает набором свойств, характерных для обоих этих компонентов. А за расположение надписи относительно текстового поля отвечает свойство LabelPosition, которое может принимать значения lpAbove, lpBelow, lpLeft и lpRigth для расположения подписи сверху, снизу, слева или справа от редактора.

Еще одним компонентом, совмещающим в себе несколько, является CheckListBox, или список опций. Он основан на обычном списке (ListBox), однако каждый его элемент, фактически, представляет собой компонент типа CheckBox. Заполняется значениями такой список опций точно так же как обычный список, а проверить состояние того или иного переключателя можно при помощи специального свойства Checked, являющегося массивом, хранящим данные типа Boolean. При этом каждый элемент этого массива соответствует строке списка, а его значение (ложь или истина) указывает на то, отмечена опция или нет.

Компонент ColorBox является элементом интерфейса, предназначенным специально для выбора цвета. Цвет, установленный по умолчанию назначается через свойство DefaultColorColor, а выбранный пользователем цвет доступен во время выполнения через свойство Selected. За доступные цвета и формат вывода названий цветов отвечает свойство Style, имеющее следующий набор флагов:

- cbStandardColors – Список будет включать в себя 16 основных цветов – черный, белый, красный, синий и т.д.;
- cbExtendedColors – Список будет включать в себя несколько дополнительных оттенков, например, светло-зеленый, кремовый и т.д.;
- cbSystemColors – Список будет включать в себя набор цветов системной палитры Windows. Сами цвета при этом зависят от настроек системы;
- cbIncludeNone – Список будет включать в себя вариант «None», т.е. без цвета. Эта опция доступна, только если установлен флаг cbSystemColors;
- cbIncludeDefault – Список будет включать в себя вариант «Default», или «цвет по умолчанию». Эта опция доступна, только если установлен флаг cbSystemColors;
- cbCustomColor – Первым в списке будет вариант Custom (произвольный цвет). Если ползователь выберет этот пункт, то откроется стандартный диалог выбора цвета;
- cbPrettyNames – Делает названия цветов более удобочитаемыми, убирая из названия цвета префикс cl.

Таким образом, компонент ColorBox является весьма удобным элементом интерфейса в тех случаях, когда необходимо дать пользователю возможность быстрого и достаточно наглядного выбора цвета.

Еще один, довольно интересный компонент, который находится на закладке дополнительных компонент – это Chart. С его помощью можно создавать графики и диаграммы самого разнообразного вида. Для этого достаточно поместить компонент на форму и дважды щелкнуть по нему, чтобы открылось окно редактирования диаграммы (рис. 16.3).

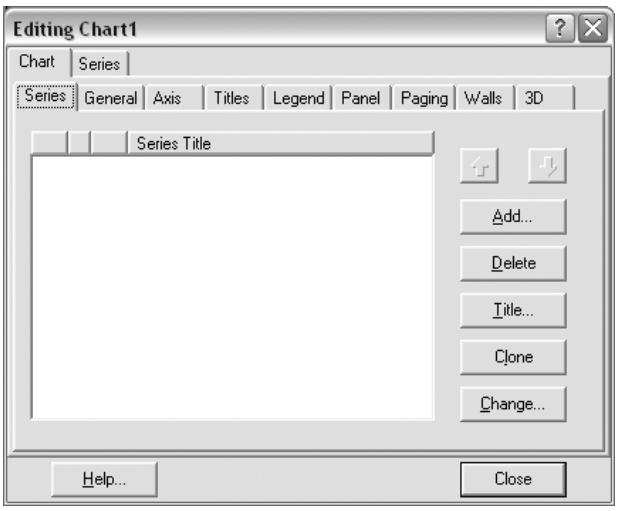

Рис. 16.3. Редактор диаграмм

Чтобы создать диаграмму, следует нажать на кнопку Add, после чего в открывшемся окне галереи выбрать нужный тип диаграммы. Сразу после этого на панели отобразится пример выбранной диаграммы, заполненной произвольными значениями. Однако если сразу же запустить программу на выполнение, панель окажется пустой, поскольку примеры видны только на этапе проектирования. Поэтому следует прибегнуть к программному коду, чтобы заполнить диаграмму значениями. Наиболее простой и наглядный метод – это использование метода Add для свойства Series. Например, чтобы заполнить круговую диаграмму, отображающую доли 6 различных значений, можно использовать следующий код:

```
Chart1.Series[0].Add(135,'Домашние сети');
Chart1.Series[0].Add(123,'ADSL');
Chart1.Series[0].Add(99,'Коммутируемые');
Chart1.Series[0].Add(52,'Кабельные сети');
Chart1.Series[0].Add(19,'GPRS');
Chart1.Series[0].Add(17,'Другие типы');
```

Для выбора заголовка диаграммы используется свойство Text.Title, являющееся списком строк. В типичном случае следует лишь изменить 0-й элемент этого списка, который по умолчанию имеет значение «TCart»:

```
Chart1.Title.Text[0]:='Типы подключения к Интернету в Москве';
```

Если разместить на форму кнопку, а приведенный выше код использовать в качестве обработчика события OnClick, то при нажатии не нее в области построения будет выводиться соответствующая диаграмма (рис. 16.4).

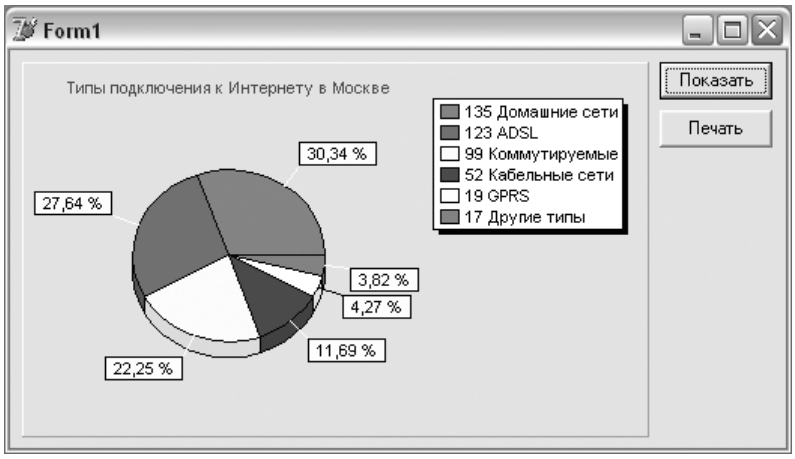

Рис. 16.4. Построенная диаграмма

Так же возможна печать диаграммы, для чего у компонента Chart имеется метод Print, так что выводить построенные диаграммы на печать чрезвычайно просто:

```
Chart1.Print;
```

В целом же этот компонент имеет множество настроек, позволяющих манипулировать видом диаграмм, вариантами подписей к значениям и т.д.

Компоненты ActiveX

Помимо компонентов VCL, Delphi может работать с инородными компонентами, созданными на основе любых иных программ, используя технологию ActiveX. На закладке ActiveX присутствует насколько таких компонент:

- Chartfx – Этот компонент служит для создания диаграмм, причем процесс создания может производиться непосредственно пользователем;
- VSSpell – Служит для осуществления орфографической проверки;
- F1Book – Готовая электронная таблица;
- VtChart – Обеспечивает построение двумерных и трехмерных диаграмм.

На самом деле, использовать все эти компоненты вряд ли имеет смысл, поскольку, например, вместо тех же ChartFx и VtChart можно использовать «родной» VCL-компонент Chart, а для двух оставшихся так же можно найти замену, обратившись к каталогам VCL-компонент в Интернете.

Основная проблема нелюбви разработчиков к ActiveX заключается в том, что эти компоненты, по сути, представляют собой отдельные приложения, которые не только придется включать в комплект поставки разрабатываемой собственной программы, но еще и заботиться об их правильной установке.

Вместе с тем, существует, по крайней мере, одно ActiveX-приложение, для которого, с одной стороны, вряд ли удастся найти подходящую замену, написанную на Delphi, а с другой, которое не требуется устанавливать, поскольку оно и так уже имеется практически у 100% пользователей Windows. Это компонент WebBrowser, располо-

женный на закладке Internet. Он представляет собой интерфейс к MS Internet Explorer, и позволяет использовать все основные возможности браузера в создаваемом приложении.

ПРИМЕЧАНИЕ

> На самом деле, здесь тоже не все гладко, поскольку у разных пользователей будут разные версии MSIE. И хотя данный компонент может работать с любой версией, начиная с 5.0, отличия в самом «движке» MSIE иногда могут служить причиной непредсказуемого результата при отображении выводимой через этот компонент информации.

Чтобы дополнить любое приложение средствами просмотра HTML-документов и даже навигации в Интернете, достаточно поместить компонент WebBrowser на форму. Основным методом WebBrowser можно считать Navigate – именно он загружает указанную ссылку. Для примера создадим новое приложение, поместим на его главную форму панель, на которой, в свою очередь, должны будут разместиться адресная строка (назовем ее AddressEd) и кнопка перехода (NavBtn). Для панели установим выравнивание в alTop, после чего поместим на форму компонент WebBrowser и установим для него выравнивание в alClient, а имя – в IEWb. Теперь для события OnClick кнопки напишем единственную строку кода:

```
IEWb.Navigate(AddressEd.Text);
```

Вот, собственно, и все, мы получили практически полноценный браузер минималиста (рис. 16.5).

Рис. 16.5. Программа-браузер из 1 строки набранного кода

Для удобства использования можно установить свойство Default кнопки NavBtn в истину, чтобы переход на набранную в адресной строке страницу осуществлялся непосредственно при нажатии на клавишу Enter (см. пример в Demo\Part3\Megabrws).

Установка компонентов

Как ни широка коллекция VCL, иногда может возникать необходимость в использовании дополнительных компонентов. Главным условием является поддержка компонентом той же версии Delphi, что вы используете. В принципе, если компонента по-

ставляется вместе с исходными кодами (а другие, честно говоря, лучше вообще не использовать), то есть большая вероятность того, что она заработает и в другой, версии Delphi, особенно если эта версия – более новая. Такая совместимость «сверху вниз» характерна для любого программного обеспечения, впрочем, для Delphi 5-7, как правило, подходят компоненты, написанные для Delphi 4, особенно если они являются самодостаточными, т.е. не интегрируются в VCL, а работают сами по себе (например, тот же Chart). Несколько сложнее дело обстоит с наборами (коллекциями) компонент, поскольку они используют специальные установочные файлы, формат которых может подвергаться некоторым изменениям от версии к версии Delphi.

Так или иначе, попробуем расширить VCL путем установки одного простого компонента. Для примера обратимся к компоненту WebLabel, который находится в каталоге Demo\Part3\WebLabel. Данный компонент представляет собой специальную метку, которая отличается от стандартного компонента Label тем, что имеет дополнительное свойство URL, в котором задается web-адрес. Указанный в URL адрес должен будет открыться браузером при щелчке пользователя по метке. Для установки компонента рекомендуется сначала закрыть проект в Delphi (File → Close All), после чего открыть диалоговое окно установки компонента (Component → Install Component).

При помощи диалога Install Component можно устанавливать единичные компоненты как в существующие пакеты (Packages), так и в новые. Чтобы не нарушать без надобности имеющиеся стандартные пакеты, воспользуемся возможностью создания нового пакета, для чего выберем закладку «Into new package». В ней заполним пустые поля, а именно Unit file name – имя файла компонента, Package file name – имя файла пакета и Package Description – описание пакета (рис. 16.6).

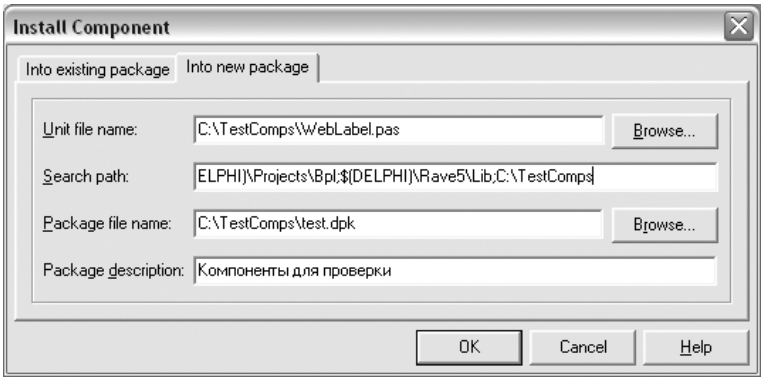

Рис. 16.6. Диалог установки компонента в новый пакет

При этом полезно предварительно создать отдельный каталог, в котором можно будет размещать все компоненты данного пакета. А для того, чтобы компилятор мог найти нужные файлы, этот каталог следует добавить к пути поиска (Search path), через точку с запятой после последнего указанного в нем каталога.

После того, как вы нажмете на кнопку OK, Delphi запросит подтверждение на создание и установку пакета, с чем надо будет согласиться. После этого пакет будет собран, а компонент – добавлен на одну из закладок палитры, в данном случае – на Samples, он там будет последним. После этого следует сохранить проект пакета и можно приступать к использованию компонента, для чего достаточно создать новое прило-

жение, поместить компонент на форму. Затем можно изменить значения свойств Caption и URL, после чего остается запустить программу и щелкнуть по надписи. В результате откроется программа просмотра Интернета и загрузит указанный адрес.

Если же устанавливать компонент в уже существующий пакет, то после диалога Install Component откроется окно пакета, в котором вам будет надо сначала нажать на кнопку компиляции (Compile), а затем – установки (Install). Конечный результат будет тем же: новый компонент добавится на палитру компонентов.

В том же случае, если устанавливается сразу несколько компонентов, то они, как правило, уже объединены в пакет. В этом случае пакет и устанавливают – для этого в диалоге открытbя файла (File → Open) из списка типов файлов надо будет выбрать Delphi Package, после чего найти и выбрать нужный файл пакета. После этого откроется окно проекта пакета, в котором, как и в случае установки единичного компонента, надо будет выполнить компиляцию и установку.

СОВЕТ

> В некоторых случаях важен порядок установки пакетов или иные особенности их интеграции с VCL. Поэтому прежде, чем устанавливать компоненты, всегда внимательно читайте поставляемые вместе с ними файлы read me и иную документацию.

В завершение темы дополнительных компонент следует отметить, что без лишней надобности лучше все-таки обходиться стандартными средствами, поскольку при большом количестве компонент от разных разработчиков могут возникать конфликты имен и т.д. Кроме того, если компонент поставляется без исходного кода, то вы будете привязаны к текущей версии Delphi, поскольку такие компоненты поставляются уже предварительно откомпилированными и могут быть использованы только совместно с той же версией компилятора.

Часть IV. Базы данных

Теория баз данных

Использование и разработка баз данных (БД) является одним из основных направлений программирования. Спектр применения БД чрезвычайно широк: практически, каждое предприятие имеет собственные базы данных, в которых, как минимум, производится бухгалтерский и складской учет, а в более крупных компаниях при помощи БД обслуживается весь цикл функционирования бизнеса. Создание БД и приложений для работы с ними – одно из основных предназначений среды Delphi.

Введение в базы данных

Прежде всего, база данных обеспечивает хранение информации и предоставляет средства для доступа к ней. Для создания, ведения и использования БД используют системы управления базами данных – СУБД.

СУБД бывают персональными и многопользовательскими. Наиболее известными локальными СУБД являются Access, FoxPro, Paradox и DBASE, а многопользовательскими – DB/2, Oracle, MS SQL Server и Interbase. Что касается Delphi, то, хотя ее и нельзя назвать СУБД как таковой, однако ее возможности по работе с БД настолько широки, что зачастую она превосходит по ним специализированные СУБД.

При этом базы данных, как и системы, их обслуживающие, подразделяются на 2 категории: локальные и клиент-серверные. Соответственно, и приложения, используемые для работы с теми или иными БД, будут либо локальными, либо клиент-серверными. Основное отличие клиент-серверных приложений состоит в том, что они могут работать с удаленными БД, в то время как локальные – только с теми, что находятся на этом же ПК (максимум – в пределах доступного сетевого диска). При этом локальные приложения так же называются одноуровневыми приложениями, а клиент-серверные бывают двух- или многоуровневыми, при этом, в любом случае, в них различают клиентскую и серверную части.

ПРИМЕЧАНИЕ

> На самом деле, для работы клиент-серверного приложения вовсе не обязательно использовать 2 компьютера, один ПК может иметь обе части приложения. Важно лишь, что к серверной части могут подключиться другие ПК.

Фактически, классификация СУБД на персональные и многопользовательские совпадает с разделением БД на локальные и клиент-серверные. Т.е. персональные СУБД, как правило, работают с локальными БД, а многопользовательские – с клиент-серверными. И действительно, например Access является типичной персональной СУБД, работающей с локальной базой. При этом потенциальная возможность обслуживания запросов от нескольких клиентов (для случая размещения БД на файл-сервере) не делает подобную СУБД истинно многопользовательской.

Но существует еще одни критерий классификации БД – по организации хранения данных, по которому они подразделяются на 3 принципиально разных типа. Это реляционные БД, иерархические БД и сетевые БД. Практически все современные БД

относятся к реляционным, концепция которых была разработана на рубеже 60х-70х годов компанией IBM.

Все уже названные СУБД предназначены для работы именно с реляционными базами данных. Особенности реляционных БД сводятся к следующим ключевым моментам:

- Все данные хранятся в таблицах, состоящих из столбцов и рядов;
- Связи между данными в различных таблицах организованы не явно, а по совпадению каких-либо ключевых значений (или по отношениям – relation);
- Каждый столбец имеет собственное имя и содержит данные только одного, явно заданного типа;
- Порядок столбцов и типы значений определяются в момент создания таблицы. При этом любая таблица должна иметь хотя бы 1 столбец;
- Ряды значений никак не упорядочены, но их можно упорядочить при обработке запроса;
- Запросы к реляционным БД возвращают результат в виде таблицы, которая, в свою очередь, так же может быть использована для запроса.

В целом к достоинствам реляционной модели данных относят простоту и удобство их реализации, а так же гибкость структуры. Из недостатков можно выделить, пожалуй, только возможность потери целостности данных, что вызвано отсутствием жестких связей между данными таблиц. В этой книге мы будем рассматривать только реляционные базы данных.

Таблицы

Таблицы, образующие БД, физически расположены в определенном каталоге на жестком диске. Обычно они хранятся в отдельных файлах, каждый из которых представляет собой отдельную таблицу или файл с дополнительной информацией (с ключами, индексами и т.п.) по таблице. В этом случае имя файла с таблицей, содержащей сами данные, совпадает с именем таблицы, заданным при ее создании. Остальные файлы, имеющие отношение к этой таблице, получают имена автоматически, при этом имя файла у них будет отличаться только расширением. В то же время, в некоторых СУБД принят режим хранения всех данных в одном файле, например, так устроено хранение данных в Access и в Interbase.

Каждая таблица БД представляет собой некоторое подобие электронной таблицы, наподобие тех, что используются в Excel. Но при этом имеется разница, как в реализации, так и в терминологии: В частности, столбцы таблицы БД называются полями, а ряды (строки) – записями. Каждое поле в таблице должно иметь уникальное имя. Так же, как уже отмечалось, в реляционных БД для каждого поля должен быть жестко задан тип данных, например, строковой, целочисленный или вещественный, и записи, хранящиеся в таблице, могут иметь данные только того типа, что предусмотрены полями.

Конкретный перечень поддерживаемых типов данных зависит от применяемой СУБД, хотя все они поддерживают, как минимум, все простые типы данных, принятых в Delphi, а именно: строки, целые числа, вещественные числа, булевы значения, а так же тип даты и времени. Еще одним часто встречающимся в БД типом данных

является бинарный объект (Blob, Binary), предназначенный для хранения практически любой информации, будь то форматированный текст, графическое изображение, или вообще данные файла произвольного типа. При этом, как правило, в самом файле табличных данных хранятся лишь ссылки на подобные объекты, которые «упаковываются» в отдельный файл, связанный с данной таблицей.

Для оптимизации размеров БД обычно предусмотрено по нескольку типов целых чисел, как минимум, от 1- до 4-байтовых – Byte, Shortint (Word) и Longint (Double Word). Что касается строк, то их максимальная длина обычно так же лимитируется. Многие современные СУБД так же позволяют работать с полями переменной длины, это относится как к строковым, так и к бинарным типам.

Таким образом, основой каждой таблицы является описание ее полей. А учитывая то, что любая таблица должна иметь хотя бы одно поле, то при создании таблицы, помимо ее имени, указывают и ее структуру, с минимальном случае состоящую из описания единственного поля. В целом же структура таблицы может включать в себя следующие составные части:

- описание полей;
- индексы;
- ключи;
- ограничения на значения;
- пароли и права доступа.

Разумеется, что поддержка той или иной возможности зависит от используемой СУБД, в частности dBase не работает с ключами. Впрочем, ключи и индексы будут подробно рассмотрены чуть позже.

После создания таблицы можно реструктуризировать, меняя количество, имена и типы полей. Однако на изменения полей налагается ряд ограничений, в частности, можно только преобразовывать типы между родственными типами, например, можно изменить тип данных с короткого целого на длинное целое или на вещественное. Однако обратная операция, скорее всего, будет невозможна, равно как и попытка изменения строкового значения на числовое и т.д. Поэтому очень важно изначально разработать как можно более точную и правильную структуру. Для этих целей следует предварительно произвести анализ данных, которые будут храниться как в БД в целом, так и в создаваемой таблице в частности.

Другим изменением, доступным после создания таблицы, является возможность ее переименования. Здесь следует отметить, что если поля в пределах каждой таблицы должны иметь уникальное имя, то в пределах БД каждая таблица так же должна иметь собственное, уникальное название. Важно отметить, что даже если выбранная СУБД хранит все данные в отдельных файлах, переименование таблиц следует делать не путем непосредственного переименования файла с данными (например, в Проводнике Windows), а средствами самой БД, т.к. надлежащим образом должны быть переименованы и все связанные с данной таблицей файлы.

Наконец, таблицу можно просто удалить. Пожалуй, это единственная операция, которую можно сделать, удалив файл непосредственно, воспользовавшись средствами файловой системы, без участия СУБД. Но, разумеется, это возможно лишь в том слу-

чае, если используемая СУБД хранит каждую таблицу в отдельном файле. В целом же любые операции лучше производить все-таки средствами самой СУБД, поскольку это послужит некоторой гарантией тому, что операция пройдет без ущерба для остальных данных.

Ключи и индексы

Поскольку реляционные БД не имеют жесткой связи между данными, хранящимися в разных таблицах, данная задача возлагается на разработчика БД. Так же остается открытым вопрос упорядочивания данных. Для данных целей предусмотрены, соответственно, ключи и индексы. При этом ключ, иногда так же называемый первичным индексом, служит для однозначной идентификации каждой отдельной записи в таблице. Ключи бывают простыми, состоящими из одного поля, и составными, объединяющими несколько полей. То поле или поля, по которым построен ключ, называются ключевыми полями таблицы. При этом в каждой таблице может быть определен только один ключ.

Помимо однозначной идентификации записи и установки связей между разными таблицами, ключ так же обеспечивает ускорение обращения к БД, включая такие операции, как поиск и сортировка данных.

Информация о ключе обычно хранится в отдельном файле, причем хранятся они, в отличие от самих записей, в упорядоченном виде. Для каждого значения ключа имеется уникальная ссылка, указывающая на расположение соответствующих этому значению данных в основной таблице. Поэтому при поиске информации по ключу вместо последовательного просмотра данных самой таблицы, СУБД просматривает значения ключей, и сразу переходит в нужное место основной таблицы. Фактически, ключ представляет собой специальную таблицу, служащую для обеспечения быстрого доступа к данным основной таблицы.

Обратной стороной использования ключей является увеличение размеров БД из-за необходимости хранения информации о ключах. При этом размер, занимаемый ключами, зависит не только от количества записей, но и от числа полей, образующих ключ и от их типа данных, поскольку в ключевом файле хранятся не только ссылки, но и сами значения ключевых полей.

Для ключей существуют определенные правила. Прежде всего, ключ должен быть уникальным. При этом уникальность может быть основана как на заведомо неповторяющихся значениях одного поля, так и на заведомо невозможном повторении сразу всех значений, используемых в составном ключе. Скажем, если у нас имеется таблица сотрудников, то в качестве единственного ключевого поля можно использовать ИНН, поскольку он заведомо уникален для каждого человека, за чем зорко следят налоговые органы. Если же такой информации не предусмотрено, то ключ можно установить, скажем, по 2 полям, скажем, по имени и дате рождения, поскольку использование только одного из этих полей мало приемлемо на случай совпадения имен или, тем более, дат рождения. Вероятность же совпадения обоих значений сразу, фактически, нулевая (если только вы не делаете БД для переписи населения).

Другим правилом является принцип разумной достаточности и неизбыточности. В частности, для составного ключа не должны быть использованы поля, удаление или изменение которых может привести к нарушению уникальности ключа. В то же время, для оптимизации размеров и быстродействия БД ключ не должен содержать

лишнюю информацию. В частности, для рассмотренного примера с таблицей сотрудников нет необходимости использовать более 1 поля, если ключ построен по ИНН, или задействовать, скажем, еще и поле с информацией о поле сотрудника, когда уже задействовано имя.

Впрочем, в ряде случаев использование каких-либо полей, предусмотренных необходимой структурой БД, в качестве ключевых неоправданно. Допустим, у нас имеется таблица номенклатуры товаров, имеющую поля типа названия товара, его параметров (скажем, цвет, фасон и размер) и цены. Допустим, в нем будет запись типа «Рубашка х/б, белая, длинный рукав, 46-48, 300р.». В данном случае создание индекса даже по всем полям, в общем-то, не гарантирует его уникальности, не говоря уже об его эффективности. В такой ситуации обычно добавляют дополнительное поле, задающее заведомо уникальный идентификатор товара – ID, или артикул, и назначают индексом именно его. В качестве подобных полей часто используют поле специального автоинкрементного типа, предусмотренного в некоторых СУБД, например, в Paradox. В таком случае при создании новой записи значение в это поле заносится автоматически самой СУБД, при этом оно будет на 1 больше предыдущего. Если же автоинкрементного типа не предусмотрено, то используют специальные триггеры и генераторы (в Interbase), или создают код каким-либо иным программным способом.

Помимо рассмотренных ключей – первичных индексов, существуют обычные индексы. В отличие от первичных, они могут содержать повторяющиеся значения. Кроме того, в таблице может быть несколько индексов. Подобно ключам, индексы могут быть построены по одному или по нескольким полям таблицы. Те поля, по которым построен индекс, называются индексными, а сам процесс построения индекса называют индексированием таблицы.

Индексы предназначены для обеспечения ускоренного доступа к данным, включая поиск и сортировку. Так же индексы могут быть использованы для установления связей между таблицами. В последнем случае индекс обычно используется совместно с ключом из другой таблицы.

ПРИМЕЧАНИЕ

> Следует учитывать, что сортировка данных в некоторых случаях (например, при использовании объекта Table) возможна только по индексированным полям. Вместе с тем, если даже использовать другие способы для доступа к данным, сортировка по проиндексированным полям происходит быстрее.

Как уже было отмечено, для каждой таблицы можно определить несколько индексов. При этом в один момент времени любой из них можно выбрать в качестве активного, т.е. того, по которому будет производиться выборка и сортировка данных таблицы.

В сумме использование ключей и индексов позволяет эффективно реализовывать следующее:

- добиться однозначной идентификации записей и избегать дублирования ключевых значений;
- устанавливать связи между отдельными таблицами;
- ускорять (а в некоторых случаях – вообще делать возможной) сортировку, а так же поиск в таблицах.

Таким образом, скорость работы БД, равно как и целостность хранимых в ней данных во многом обеспечиваются как раз правильным подбором ключей и индексов в таблицах.

Организация данных и связь между таблицами

При рассмотрении ключей и индексов мы много раз упоминали про организацию связи между различными таблицами в рамках БД. И хотя теоретически БД может состоять из одной единственной таблицы, содержащей лишь имена и даты рождения сотрудников, на практике СУБД для подобных целей не применяются: такую информацию можно хранить в виде электронной таблицы, а то и вообще в обычном текстовом файле. Поэтому рассмотрим более типичную ситуацию, когда БД состоит из нескольких взаимосвязанных таблиц.

Организация отношений между таблицами называется связыванием, или соединением таблиц и является одним из ключевых моментов разработки БД. Связи задаются как при создании таблиц (назначение ключей и индексов), так и в процессе разработки и даже работы приложения, для чего используются различные средства, предоставляемые СУБД.

Для связывания таблиц используют поля, называемые полями связи. Такие поля должны быть индексированными. При связывании таблиц одна из них назначается главной (Master), а другая – подчиненной (Slave, Detail). При этом индекс, отвечающий за соединение в подчиненной таблице, называют внешним ключом. Что касается главной таблицы, то ее поле, участвующее в связывании, должно быть главным индексом (разумеется, если в применяемой СУБД есть такое понятие). Типы полей, участвующих в связывании, должны совпадать, например, оба они должны быть целочисленными.

Допустим, у нас имеется БД, состоящая из 2 таблиц, в одной из которых хранится список клиентов (назовем ее CUSTOMER), а в другой – накладные, выписанные этим клиентам (BILL). В таком случае ключом первой таблицы должен быть идентификатор клиента (скажем, поле автоинкрементного типа CUST_ID), а во второй должно иметься, по возможности, индексированное поле, содержащее информацию о том, какому клиенту выдана данная накладная (поле BILL_CUST). При этом ключевым полем второй таблицы будет номер накладной (поле BILL_ID). В получившейся связи связанными полями будут CUST_ID и BILL_CUST, причем последнее будет являться внешним ключом для таблицы BILL (рис. 17.1).

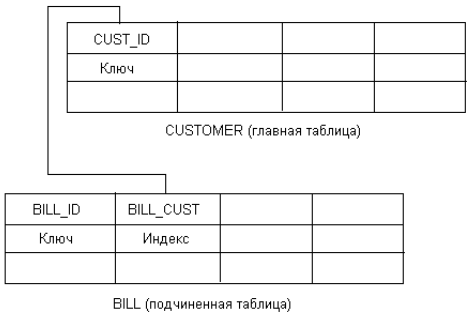

Рис. 17.1. Связи между таблицами

Связать между собой можно 2 таблицы или несколько таблиц сразу. При этом одна из таблиц обычно выступает главной, а другие – подчиненными. Такую связь обычно называют «главный-подчиненный». Вообще же, в зависимости от заданных условий, эта связь может получиться одного из 4 видов:

- «Один к одному», когда каждой записи в главной таблице соответствует 1 запись в подчиненной;
- «Один ко многим», когда каждой записи в главной таблице соответствует 0 или больше записей в подчиненной;
- «Много к одному», когда нескольким записям в главной таблице соответствует 1 в подчиненной;
- «Много ко многим», когда произвольному числу записей в главной таблице соответствует такое же неопределенное число записей в подчиненной.

Чаще всего встречается 2-й вариант, включая рассмотренный пример, где каждой записи о клиенте может соответствовать произвольное число записей в таблице накладных, включая полное их отсутствие, если клиент еще ничего не купил.

После того, как связь между таблицами установлена, выбор определенной записи в главной таблице автоматически приведет к выборке в подчиненной (рис. 17.2).

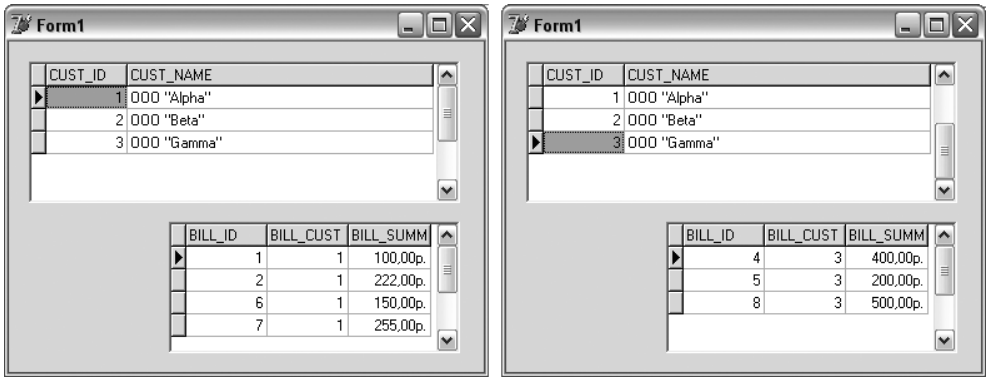

Рис. 17.2. Связь «один ко многим», связанные поля BILL_CUST и CUST_ID

Хотя технология связывания таблиц средствами IDE будет рассматриваться позже, забегая вперед, отметим, что в данном случае для таблицы BILL была выбрана Master-таблица CUSTOMER и была установлена связь через объединение полей типа BILL_CUST → CUST_ID (подробнее см. в главе 19).

Целостность БД и транзакции

При проектировании приложений БД следует учитывать взаимосвязь между таблицами. Например, для того, чтобы не допустить появления в таблице накладных «потерянных» записей, не относящихся ни к одному из клиентов, при удалении записи в таблице CUSTOMER следует выполнять проверку таблицы BILL на наличие накладных, имеющих отношение к данному клиенту и удалять их. Если же информация о счетах важна, скажем, для ведения статистики продаж, то следует изменять значение в поле BILL_CUST на какого-либо «клиента по умолчанию». В то же время, при до-

бавлении записи в подчиненную таблицу надо заботиться о том, чтобы во внешнем ключе было установлено верное значение, ссылающееся на главную таблицу.

Сами ограничения по установке, изменению и удалению полей в связанных таблицах могут быть наложены при создании таблиц средствами, предоставляемыми СУБД, параллельно описанию полей и индексов. В таком случае данные ограничения будут являться частью структуры таблиц, и за их выполнением будет следить СУБД. В то же время, эти ограничения могут и не являться частью структуры БД, в таком случае за их выполнением должен будет следить разработчик при создании приложения. Например, если производится попытка удалить запись из главной таблицы (скажем, из CUSTOMER), то при этом должны быть произведена проверка подчиненной таблицы (BILL). И, в зависимости от логики программы, должны будут либо удалены все записи, относящиеся к данному клиенту в таблице накладных, либо удаление клиента должны быть запрещено. В любом случае при этом полезно выдавать предупреждающее сообщение оператору, работающему с приложением БД.

Подобное поведение БД, когда действие в одной таблице порождает ряд действий в других (например, то же удаление в подчиненных таблицах), называется каскадированием. Каскадирование, как и иные описанные выше шаги, направлены на обеспечение целостности БД. При каскадировании может быть выбран различный порядок выполнения операций: например, можно сначала удалить запись в главной таблице, после чего удалить записи в подчиненной, а можно наоборот, сначала удалить записи в подчиненной, после чего – удалить в главной. Но в любом случае остается открытым вопрос обеспечения целостности БД на случай непредвиденных сбоев, как-то отключения электропитания, разрыва соединения (при работе с удаленной БД) и иных неприятностей. В таком случае может возникнуть ситуация, когда в одной таблице данные будут удалены, а в другой – нет, в результате чего целостность данных будет нарушена.

На помощь в таких ситуациях приходит использование механизма транзакций. Транзакция представляет собой выполнение последовательности операций как общего целого. При выполнении транзакции возможно 2 варианта:

- все операции успешно выполнены. В таком случае транзакция считается успешной, изменения записываются во все таблицы и БД в результате переходит из одного целостного состояния в другое;
- одна или более операций завершена неудачно. В таком случае транзакция считается неуспешной и результаты всех(!) операций, образующих данную транзакцию, отменяются, и БД остается в целостном состоянии.

Использование транзакций крайне необходимо в таких случаях, когда выполняются сложные каскады операций, а так же при многопользовательском доступе к БД.

Транзакции бывают неявными, когда она стартует и завершается автоматически, за что отвечают механизмы СУБД, а так же явными, когда за их выполнение отвечает программный код создаваемого приложения БД. Например, для переноса данных из одной таблицы в другую, можно начать транзакцию при считывании и удалении данных из одной таблицы и завершить – после внесения записи в другую. В противном случае (без использования механизма транзакций) мы могли бы столкнуться с такой ситуацией, что в данных не окажется ни в одной из таблиц, или, наоборот, часть данных окажется продублированной в обоих.

В дополнение к вопросу целостности и транзакций следует упомянуть про такое понятие, как бизнес-правила. Бизнес-правила предназначены для выполнения целого ряда действий, облегчающих работу с БД, в том числе и поддержание базы в целостном состоянии:

- задание значений по умолчанию для полей таблиц;
- запрет пустого значения;
- требование уникального значения;
- определение допустимого диапазона значений;
- ограничение ссылочной целостности.

Обычно бизнес-правила задаются при создании таблиц, хотя ничего не мешает установить их, или дополнить на программном уровне уже в процессе разработки приложения.

Язык SQL

Рассказывая о реляционных базах данных, невозможно не упомянуть о таком инструменте, как SQL – Structured Query Language (язык структурных запросов). Язык SQL предназначен для создания запросов к базам данных и существенно отличается от других языков программирования, в том числе от Delphi.

Прежде всего, SQL не является процедурным языком (хотя в промышленных СУБД и реализована возможность создания процедур на SQL). Каждый запрос определяет, что нужно сделать с данными и выполняется сам по себе, а не является последовательностью инструкций. Например, простейший запрос, который выведет все содержимое таблицы BILL, будет выглядеть следующим образом:

```
SELECT * FROM BILL
```

При этом все операции по интерпретации и выполнению запроса выполняются СУБД, а приложению остается лишь отсылать запросы и отображать их результаты. В данном случае был использован один из основных операторов SQL – SELECT. Всего в SQL имеется 3 группы операторов:

1. Операторы управления данными (Data Manipulation Language, DML), служащие для выполнения поиска, удаления, изменения и сохранения данных. Помимо SELECT, к ним относятся UPDATE, INSERT и DELETE.
2. Операторы определения данных (Data Definition Language, DDL), которые используются для создания объектов БД и изменения их структуры. Это операторы CREATE TABLE, CREATE VIEW, CREATE DOMAIN и т.д.
3. Операторы контроля данных (Data Control Statements, DCS), использующиеся для контроля прав доступа к данным – GRANT и REVOKE.

Кроме них, в SQL имеются дополнительные операторы, служащие для управления транзакциями, сеансами, соединениями и т.д. Набор операторов SQL, как и их синтаксис может варьироваться в зависимости от применяемой СУБД. И хотя практически все СУБД поддерживают основной набор и синтаксис операторов, определенных в действующем стандарте ANSI SQL, они могут иметь собственные расширения языка и особенности реализации тех или иных стандартных процедур.

ПРИМЕЧАНИЕ

> К сожалению, в рамках настоящего издания мы не сможем детально изучить язык SQL, поскольку он сам по себе представляет очень емкую тему. И если вы планируете создавать клиент-серверные БД, то он вам будет необходимо обратиться к дополнительным источникам информации, поскольку работа с такими БД строится именно на основе SQL.

Еще одним отличием SQL от других языков программирования является его тройственная логика. В частности, помимо значений False и True (ложь и истина), логические переменные могут содержать значение Unknown (неизвестно), что соответствует пустой ячейке таблицы или специальному значению Null.

И хотя те СУБД, с которыми мы, преимущественно, будем сталкиваться по ходу дальнейшего изучения баз данных, не являются основанными на SQL, среда Delphi позволяет работать с ними не только собственными средствами, но так же и при помощи языка SQL, пусть и с некоторыми ограничениями. В то же время основная масса примеров будет обходиться без SQL-запросов вообще, опираясь лишь на средства локальных СУБД и на инструментарий, имеющийся в Delphi.

Delphi и базы данных

Ознакомившись с основными терминами и понятиями БД, можно перейти к рассмотрению средств, имеющихся в распоряжении разработчика, создающего приложения БД при помощи Delphi. Средства Delphi, предназначенные для работы с БД, можно разделить на 2 категории: инструментарий и компоненты. Инструментарий включает в себя рад дополнительных программ, входящих в поставку Delphi и предназначенных для работы с БД. Что касается компонентов, то в VCL имеется множество специализированных компонент, предназначенных для работы с БД.

Типы БД в Delphi

Хотя Delphi и не является СУБД в классическом понимании этого термина, в частности, у нее нет собственного формата баз данных, она поддерживает как собственные сразу несколько форматов, в частности, dBase и Paradox.

Таблицы dBase являются одним из первых форматов БД, появившихся на ПК, благодаря чему с ними могут взаимодействовать многие СУБД – если и не на прямую, то по крайней мере, имея возможность импортировать и экспортировать данные в этом формате. Более того, с некоторыми оговорками, их можно просматривать и редактировать даже в электронных таблицах, например, в Excel.

Формат dBase является достаточно простым и может использовать для хранения каждой таблицы до 3 файлов:

- dbf – основная таблица с данными;
- dbt – BLOB-данные (этот файл появляется, если в таблице предусмотрены соответствующие поля);
- mdx – файл с индексами.

Таким образом, dBase – это достаточно простой и универсальный формат таблиц. Это о преимуществах, из недостатков же следует отметить такие, как ограничения на

имена полей в таблицах: максимальная длина – всего 10 символов, допустимы только латинские буквы и цифры. Сходные ограничения налагаются на имена самих таблиц, при этом, в лучших традициях 16-битных систем, длина имени таблицы не должна превышать 8 символов (хотя это ограничение, в общем-то, можно игнорировать при работе в Delphi под 32-разрядными системами). Но куда более существенными недостатками являются отсутствие автоматического контроля целостности данных или связей, прав доступа, а так же ограничения на поддерживаемые типы данных, ключи, индексы и т.д. Все это делает данный формат недостаточно конкурентоспособным.

Другой формат, Paradox, появился позже и, пожалуй, является одним из наиболее развитых форматов, применяемых в локальных БД. Основными отличиями этого формата от dBase являются:

- возможность использования практически любых символов для названий полей, а так же лимит в 25 символов, а не в 10;
- большее количество поддерживаемых типов данных, включая автоинкрементный тип;
- поддержка контроля целостности данных и возможность организации проверки вводимых данных;
- возможность защиты таблиц паролем и определения прав доступа к данным.

Благодаря таким достоинствам, формат Paradox используется гораздо чаще. Кроме того, большой набор типов данных позволяет эффективно подбирать нужный тип для хранения данных. Полный перечень поддерживаемых Paradox типов данных и их символьные обозначения в программе Database Desktop приведены в таблице 18.1.

Таблица 18.1. Типы данных в Paradox 7

Тип	Обозначение	Описание
Alpha	A	Строка длиной не более 255 символов (аналог ShortString)
Number	N	Число с плавающей точкой (аналог Double)
Money	$	То же, что и Number, но при выводе сопровождается обозначением денежного знака
Short	S	Малое целое (аналог SmallInt)
LongInteger	I	Целое (аналог LongInt)
BCD	#	Число в двоично-десятичном формате
Date	D	Дата в диапазоне от 1 января 9999г. до н.э. до 31 декабря 9999г.
Time	T	Время
Timestamp	@	Дата и время
Memo	M	Строка произвольной длины. Первые 240 символов хранятся в файле основной таблице, остальные – в файле с BLOB-данными
Formatted Memo	F	То же, что и Memo но для RTF-текста
Graphic	G	Графическое изображение в формате BMP, PCX, TIFF, GIF или EPS. При выводе преобразуется в BMP, хранится в файле с BLOB-данными

OLE	O	Произвольные данные, которые поддерживаются через OLE. Хранятся в файле в BLOB-данными
Logical	L	Логическое значение (TRUE или FALSE)
Autoincrement	+	Автоинкрементное поле. При добавлении новой записи значение этого поля автоматически увеличивается на 1
Binary	B	Двоичная информация (произвольная последовательность байтов). Подобно Memo, первые 240 байт хранятся в файле основной таблицы, остальные – в BLOB
Bytes	Y	Двоичная информация (произвольная последовательность байтов) длиной не более 255 символов

Что касается файлов, используемых для хранения информации, то в Paradox их может быть использовано достаточно большое количество:

- db – файл с основными данными таблицы;
- mb – файл с BLOB-данными;
- px – файл с первичным индексом (ключом);
- xg? и yg? – файлы со вторичными индексами (вместо ? будет порядковый номер индекса);
- val – файл с информацией для проверки данных и целостности ссылок.

Таким образом, количество файлов, ассоциированных с каждой таблицей, может достигать десятков, если использовать множество вторичных индексов. Пожалуй, это и есть единственный существенный недостаток формата Paradox. Но поскольку преимуществ у него гораздо больше, то в Delphi именно Paradox предлагается использовать по умолчанию, и потому этот формат называется не Paradox, а Standard.

Инструменты для работы с БД

Помимо поддержки БД в самой среде Delphi, в состав поставки Delphi входит ряд дополнительных инструментов, служащих для обеспечения работы с БД. Прежде всего, это BDE – Borland Database Engine, представляющий собой набор системных библиотек и драйверов, предназначенных для взаимодействия БД и приложений, разрабатываемых в Delphi.

Кроме того, имеется рад дополнительных приложений, вызвать которые можно самостоятельно (из программной группы Delphi), или из IDE:

- BDE Administrator – утилита для настройки различных параметров BDE, настройки драйверов баз данных, создания и удаления псевдонимов БД. Эта программа так же встраивается в системную панель управления;
- Database Desktop – программа для создания, просмотра и редактирования отдельных таблиц и запросов;
- Data Pump – программа для переноса данных между БД;
- Database Explorer – программа-проводник для иерархического просмотра и редактирования БД. В версиях Enterprise и Architect устанавливается более гибкая и функциональная версия – SQL Explorer;

- SQL Monitor – программа для отслеживания прохождения запросов к удаленным серверам БД (только Enterprise и Architect);
- SQL Builder – приложение для конструирования SQL-запросов (вызывается при обращении к соответствующим компонентам в Delphi IDE, только Enterprise и Architect);
- InterBase Server – СУБД InterBase, включая клиентскую и серверную части (только Enterprise и Architect);
- dbExpress – набор драйверов для доступа к SQL-СУБД, включая InterBase, DB2, Oracle, MSSQL и MySQL. В VCL для этих целей используется одноименный набор компонентов.

В ранних версиях Delphi вместо dbExpress использовались SQL Links, но, начиная с Delphi 7, эта система считается устаревшей. Как и другие SQL-ориентированные компоненты, dbExpress и SQL Links имеются только в версиях Enterprise и Architect.

ПРИМЕЧАНИЕ

> Очевидно, что многие инструменты, прежде всего, ориентированные на использование с промышленными БД, имеются в промышленных же (Enterprise) вариантах поставки Delphi стоимостью около 2-3 тыс. долл. Впрочем, рассмотрение подобных нюансов в этой книге мы опустим, тем более что для рассматриваемых примеров SQL-ориентированные СУБД нам не понадобятся.

Помимо перечисленных инструментов, в Delphi имеется множество компонент, имеющих самое непосредственное отношение к базам данных. Прежде всего, это группы Data Access, Data Controls и BDE. К этой же категории относятся группы ADO, InterBase, WebSnap и некоторые другие, однако все их рассматривать не представляется ни возможным, ни необходимым.

Таким образом, в состав Delphi входит все необходимое для того, чтобы создавать как локальные, так и промышленные – если рассматривать соответствующий вариант поставки Delphi – базы данных.

ВНИМАНИЕ

> Следует учитывать, что используемые драйвера СУБД, равно как и BDE в обязательном порядке должны присутствовать на тех ПК, на которых будет выполняться разрабатываемое приложение БД.

Помимо средств, входящих в поставку Delphi необходимо упомянуть и о технологиях, имеющихся для этих целей в Windows. Так, механизм доступа к данным ADO, основанный на COM обеспечивает универсальный механизм доступа к данным из приложений. Использование ADO позволяет отказаться от установки BDE и поставки дополнительных библиотек на ПК конечного пользователя разрабатываемого приложения БД. Ряд компонент, в частности, компоненты из групп dbExpress и InterBase так же позволяют работать с БД в обход BDE. Однако отказ от BDE и использование специализированных компонентов автоматически увеличивает сложность разработки приложений баз данных, поскольку BDE по отношению к различным СУБД, как и VCL по отношению Windows API, сглаживает многие острые углы, проявляющиеся при работе с БД.

BDE и BDE Administrator

Несмотря на то, что в последнее время компания Borland усиленно продвигает SQL-ориентированный подход для разработки БД и агитирует за использование соответствующих компонентов (в частности, dbExpress), BDE остается популярным средством для разработки относительно несложных БД. Такому положению вещей способствует не только удобство и простота этой технологии, но и большое количество приложений, созданных с использованием BDE, поддержку и развитие которых требуется осуществлять и сегодня.

Для настойки параметров БД, поддерживаемых BDE, а так же для создания псевдонимов (Aliases) к базам данных, используют специальную утилиту, входящую в поставку Delphi – BDE Administrator. Рабочая область этой программы представляет собой блокнот из 2 страниц – Databases и Configuration (рис. 18.1).

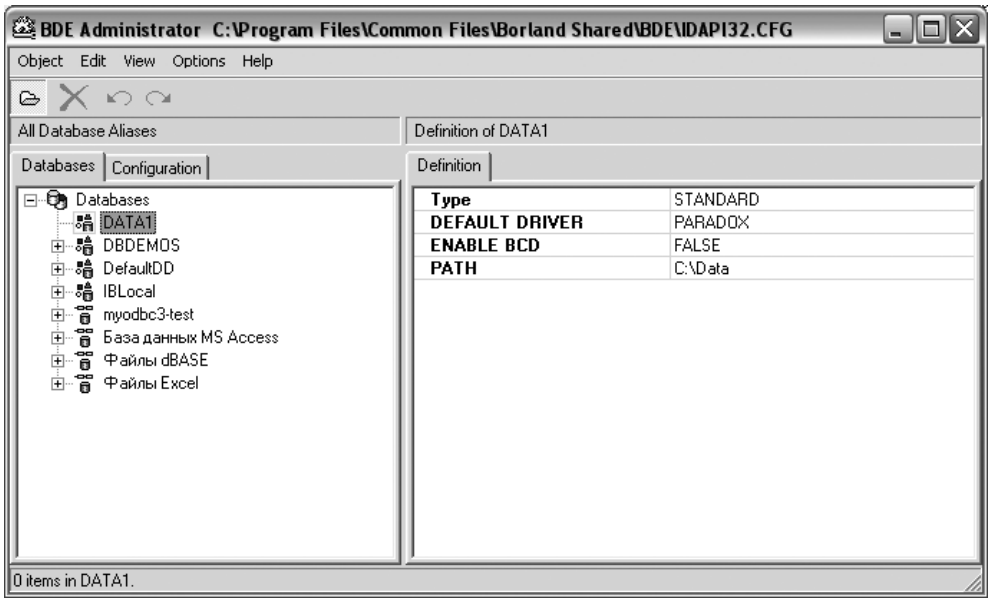

Рис. 18.1. Окно BDE Administrator с открытой страницей Databases

На странице Databases расположен список имеющихся псевдонимов к БД. Сразу после установки Delphi и BDE создается несколько псевдонимов, в частности, DBDEMOS, DefaultDD и IBLocal. А чтобы создать новый псевдоним, достаточно из меню Object выбрать пункт New. В результате откроется окно, в котором вы сможете выбрать тип СУБД. По умолчанию предлагается Standard, что обычно подразумевает использование Paradox в качестве драйвера для создаваемой БД. После этого в списке появится новый элемент, который будет назван Standard1, а справа, в области определения свойств (Definition), будут выведены все доступные для редактирования параметры. Применительно к стандартному для Delphi типу это будет собственно тип (Type), драйвер СУБД (Default driver), режим преобразования вещественных чисел (Enable BCD) и каталог, в котором расположены файлы БД (Path). Поскольку Delphi в качестве «своего» формата поддерживает не только Paradox, то в качестве драйвера можно указать так же форматы dBase, FoxPro и ASCII (текстовый формат).

После создания нового псевдонима можно изменить его название – проще всего это сделать сразу же после того, как он будет создан – пока курсор редактирования находится непосредственно на нем. Позже можно будет изменить его, выбрав пункт Rename из меню Object.

Допустим, что наш псевдоним будет называться DATA1, а в качестве пути укажем каталог Data на диске C. Таким образом, если нам в будущем понадобится обратиться к БД типа Paradox, расположенной в каталоге C:\Data, то нам будет достаточно указать лишь ее псевдоним, а остальную информацию BDE предоставит приложению автоматически.

Вторая страница BDE Administrator – Configuration предназначена для настроек как отдельных драйверов СУБД (группа Drivers), так и для общих для всех СУБД системных переменных (группа System). Драйвера, в свою очередь, подразделяются на поддерживаемые BDE напрямую (Native) и посредством ODBC (Open Database Connectivity – совместимость открытых баз данных).

Среди настроек, в первую очередь следует выделить опцию Langdriver, имеющуюся как у системных настроек (группа INIT, см. рис. 18.2), так и для каждой СУБД в отдельности. Для корректной работы с кириллицей следует установить значение «Pdox ANSI Cyrillic», соответствующее кодировке Windows-1251, как в общих настройках (INIT), так и для используемого драйвера СУБД (например, Paradox).

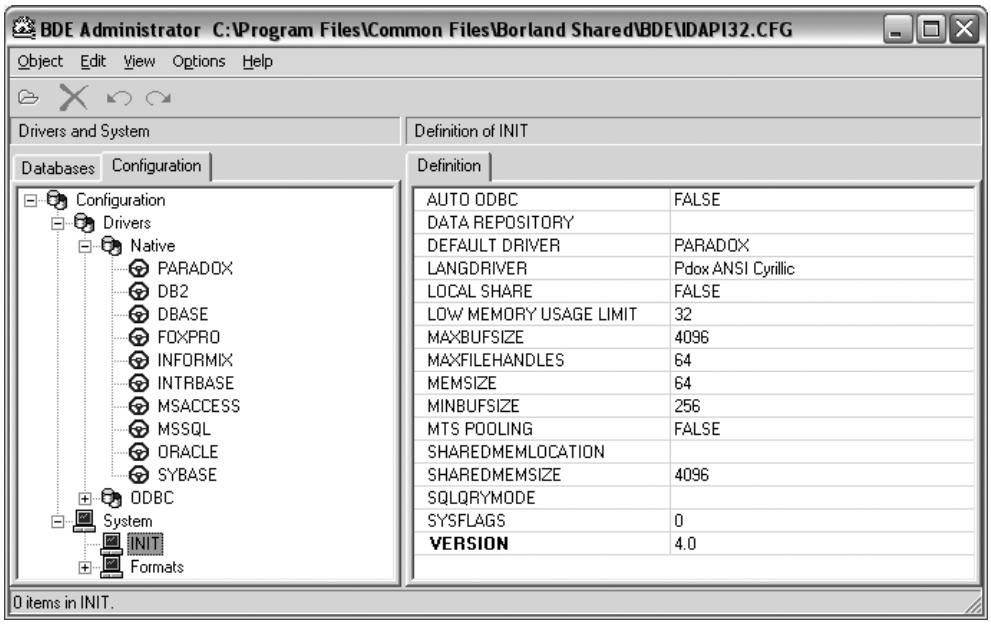

Рис. 18.2. Системные настройки в BDE Administrator

После произведения всех необходимых настроек следует перезапустить приложения, использующие BDE для того, чтобы новые параметры вступили в силу. Что касается созданных псевдонимов, то они становятся доступными сразу же после создания (вернее, сразу после того, как вы сохраните конфигурацию, нажав на кнопку Apply или выбрав соответствующий пункт из меню Object).

Создание таблиц в Database Desktop

Для работы с таблицами при создании приложения БД можно использовать приложение Database Desktop, входящее в поставку Delphi. С помощью Database Desktop можно создавать и реструктуризировать таблицы, вносить в них новые записи, а так же создавать запросы.

Для создания новой таблицы следует выбрать пункт меню File → New → Table. При этом будет предложено выбрать тип создаваемой страницы, по умолчанию предлагается формат Paradox 7, и именно его мы и будем использовать. Сразу после подтверждения выбранного типа откроется окно определения структуры таблицы (рис. 18.3), в котором и производятся все необходимые действия, связанные с созданием и определением параметров таблицы, включая ее поля, индексы, пароли, условия и ограничения на значения и для ссылочной целостности.

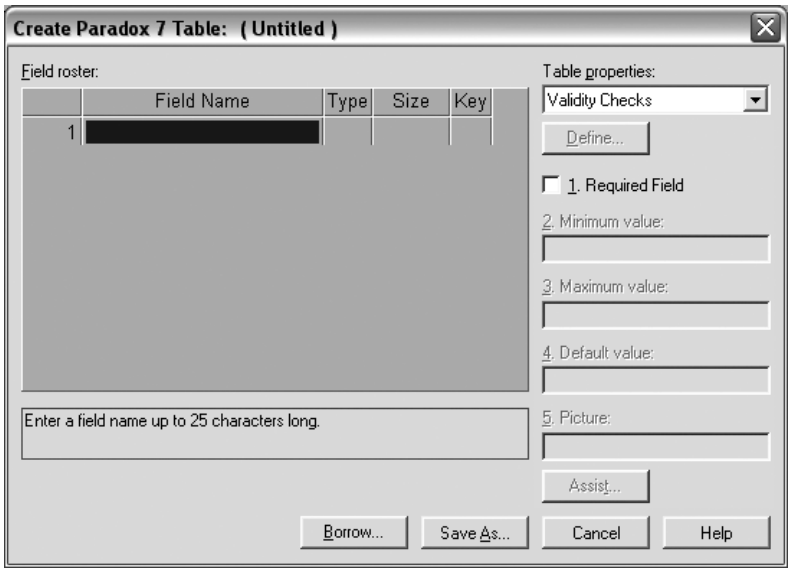

Рис. 18.3. Окно определения структуры таблицы

Как нам уже известно, программа-минимум для создания таблицы реляционной СУБД заключается в создании одного поля. Имя поля указывают в столбце Field Name, а его параметры – в Type (тип) и Size (размер). Возможные типы данных для таблиц Paradox были приведены в таблице 18.1, что касается размеров, то они актуальны, в основном, только для типов Alpha и Bytes. Для BLOB-типов данных размеры так же можно указывать, но для них этот размер будет определять не ограничение размера самих данных, а то, сколько байт должно храниться в основном файле таблицы. Наконец, в графе Key можно отметить поле или поля, являющиеся ключевыми, по которым будет создан первичный индекс. Кроме того, можно сделать целый ряд дополнительных операций, в том числе и над таблицей в целом, для чего в расположенном справа ниспадающем списке можно выбрать один из следующих пунктов:

- Validate Checks (проверка правильности) – задает ограничения значений для текущего поля;

- Table Lookup (таблица выбора) – задает правила совпадения вводимых значений со значениями из другой таблицы для поддержки целостности;
- Secondary Indexes (вторичные индексы) – определяет вторичные индексы для таблицы;
- Referential Integrity (ссылочная целостность) – определяет условия для проверки ссылочной целостности;
- Password Security (пароли) – определяет пароли для разграничения доступа к полям таблицы;
- Table Language (язык) – определяет кодовую страницу таблицы, значение по умолчанию берется из настроек BDE;
- Depended Tables (зависимые таблицы) – показывает список таблиц, зависимых от текущей таблицы по линии ссылочной целостности.

Наиболее часто возникает необходимость именно в задании индексов. В принципе, для таблиц Paradox иногда достаточно указать лишь первичный индекс, отметив нужное поле (или поля) в графе Key. Но если возникнет необходимость определить дополнительные индексы, или если за основу был взять другой формат, скажем, dBase, ключевого индекса в котором не предусмотрено, то следует выбрать из списка пункт Secondary Indexes, после чего нажать на кнопку Define. В открывшемся окне определения вторичных индексов (Define Secondary Index) из списка имеющихся в таблице полей, который отображен справа, выбирается поле или поля, по которым должен быть создан новый индекс. Внизу окна расположена группа из 4 опций, позволяющих определить тип создаваемого индекса. В частности, опция Unique указывает на то, что индекс требует уникальные значения для составляющих его полей, опция Maintained определяет, должен ли индекс обновляться при каждом изменении в таблице, а опция Case Sensitive отвечает за распознавание регистра символов в текстовых строках. Ну и оставшаяся опция – Descending – указывает на то, что сортировка индекса должна производиться не по возрастанию, как это принято по умолчанию, а по убыванию.

После того, как поля и опции будут выбраны, надо будет нажать на кнопку OK, после чего программа попросит вас ввести имя индекса. Указав имя и вновь нажав OK, вы увидите новый индекс в списке. Если понадобится изменить ранее определенный индекс, то достаточно будет выбрать его в списке и нажать на кнопку Modify. Если же какой-либо индекс окажется ненужным, то его можно удалить, нажав на Erase.

Для примера создадим простую таблицу клиентов. Она будет состоять всего из 2 полей – номера (назовем это поле CUST_ID) и имени (CUST_NAME). Для первого поля установим автоинкрементный тип (+) и сделаем его ключевым, поставив двойным щелчком мышки звездочку в графе Key. Второе поле должно быть текстовым (A), а его размер можно ограничить 40 символами, для чего в графе Size введем число 40.

Итак, когда все поля таблицы будут определены, индексы заданы, и все остальные шаги по определению структуры таблицы так же выполнены, останется лишь сохранить новую таблицу, для чего следует нажать на кнопку Save As. В диалоговом окне сохранения таблицы обратите внимание на список Alias, при помощи которого возможна быстрая навигация между имеющимися БД. Применительно к только что соз-

данной таблице в списке псевдонимов выберем DATA1, а в качестве имени файла (оно же будет именем таблицы) используем слово customer.

СОВЕТ

> Если вы создаете БД, состоящую из множества однотипных таблиц, то полезно знать, что, нажав на кнопку Borrow и указав таблицу-прототип, можно импортировать ее структуру во вновь создаваемую таблицу, после чего останется лишь внести необходимые корректировки – и новая таблица готова.

После создания и сохранения таблицы ее можно открыть для внесения записей, а так же для изменения структуры. Для этого следует воспользоваться меню File → Open → Table и выбрать таблицу в окне открытия файла. Опять-таки, как и при сохранении, здесь будет список псевдонимов, при помощи которого можно быстро выбрать каталог с нужной БД.

Открыв таблицу, вы увидите ее в форме представления данных. Если таблица не является только что созданной и уже содержит какую-либо информацию, то она так же будет видна.

ПРИМЕЧАНИЕ

> Довольно часто, особенно при работе в среде Windows 2000/XP, программа Database Desktop не отображает символы кириллицы. Для исправления этого досадного обстоятельства запустите файл cyr_fix, находящийся в каталоге Tools\DBD_Patch.

Чтобы изменять данные в таблице или вносить в нее новые записи, следует переключиться в режим редактирования, для чего можно нажать на кнопку-триггер Edit Data на панели инструментов, или выбрать одноименный пункт из меню Table. Если открыто несколько таблиц одновременно, то режим правки будет включен только для той таблицы, что была на переднем плане в момент включения этого режима. Вернуться к режиму просмотра (чтобы ненароком не повредить имеющиеся данные) можно, отжав триггер (или выбрав Table → View Data).

Откроем только что созданную таблицу Customer и заполним ее данными, для чего достаточно будет вводить данные лишь в столбец поля CUST_NAME, поскольку автоинкрементное поле CUST_ID будет заполняться автоматически. Важно отметить, что правка таблицы, как в Database Desktop, так и любым другим способом, приводит к немедленному изменению ее действительного содержания. Из этого следует, что, во-первых, операция сохранения, привычная по электронным таблицам, здесь не уместна (все сохраняется сразу после вода), а во-вторых, надо быть аккуратным, так как отменить правку не получится.

Кроме правки содержащихся в таблице данных, при помощи Database Desktop можно посмотреть структуру таблицы, а также изменить ее. Для этого в меню Table имеются пункты Info Structure (просмотр структуры) и Restructure (изменение структуры). И в том и в другом случае будет открыто уже знакомое нам окно определения структуры таблицы, с той лишь разницей, что для просмотра оно будет открыто в режиме «только для чтения».

Внося изменения в структуру таблицы, уже содержащую какие-либо данные, следует помнить, что это может привести к потере части информации. Например, если изна-

чально для какого-либо текстового поля был задан размер 100 символов, то уменьшение его размера до 60 символов приведет к тому, что информация будет «обрезана» по 60 символу, а последние 40 окажутся безвозвратно утерянными. Точно так же часть информации может быть утеряна, например, при изменении поля вещественного типа на целое, длинного целого на короткое и т.д. Правда, если подобная операция производится в Database Desktop, то при этом вам будет выдано соответствующее предупреждение. Кроме того следует учитывать, что в момент изменения структуры, никакое другое приложение не должно использовать данную таблицу.

Наконец, рассмотрим еще один вариант правки – изменение имени самой таблицы. Несмотря на то, что эту операцию, казалось бы, можно выполнить из файловой системы, переименовав файлы, используя проводник Windows, такой шаг приведет к потере связанной информации, поскольку имя таблицы хранится во всех связанных с ней файлах (фалы индексов, BLOB-данных и т.д.). Поэтому для переименования таблицы следует открыть ее для реструктуризации в Database Desktop и нажать на кнопку Save As, после чего выбрать новое имя. В результате будут корректно сохранены как основной файл данных таблицы, так и вся связанные с ним файлы индексов, метаданных и т.д. Исходные файлы при этом так же останутся на диске – вот их и можно будет удалить средствами файловой системы.

Поддержка BDE в VCL

Для создания приложений, работающих с БД через BDE, в VCL предусмотрена группа компонентов, расположенная на закладке BDE палитры компонентов. Из представленных 8 компонентов для нас сейчас наибольший интерес представляют 2, а именно Database и Table.

Компонент Database представляет собой компонент, инкапсулирующий в себе базу данных в целом. С его помощью можно контролировать наличие доступа к базе данных из приложения, следить за ходом транзакций, а так же подключаться к клиент-серверным БД. Свойства этого компонента приведены в таблице 18.2.

Таблица 18.2. Свойства компонента Database

Свойство	Тип	Описание
AliasName	String	Определяет псевдоним BDE для подключения
Connected	Boolean	Определяет, установлено или нет соединение с БД
DatabaseName	String	Определяет имя БД, ассоциированное с данным компонентом
Directory	String	Определяет рабочий каталог для БД Paradox или dBase
DriverName	String	Определяет имя драйвера BDE для данной БД
Exclusive	Boolean	Включает монопольный доступ к БД
InTransaction	Boolean	Указывает, выполняется ли в данный момент транзакция
KeepConnection	Boolean	Определяет, должно ли приложение оставаться подключенным к БД, когда активных соединений нет
Params	TStrings	Определяет список дополнительных параметров для псевдонима BDE
SessionName	String	Определяет имя сессии для данного компонента
ReadOnly	Boolean	Включает режим доступа к данным только на чтение

Пожалуй, наиболее важным свойством является DatabaseName: указав в качестве значения этого свойства один из уже определенных в BDE псевдонимов, мы получим готовый к использованию компонент. Еще одно свойство – Connected – отвечает за непосредственную установку связи с БД. Таким образом, указав в качестве значения свойства DatabaseName «DATA1», и установив свойство Connected в истину, мы подключимся к БД.

Довольно интересной особенностью компонента Database является возможность создания псевдонимов БД, действующих в рамках создаваемого приложения. Прежде всего, можно указать в качестве значения свойства Alias то же значение DATA1, выбрав его из списка, а в качестве DatabaseName указать какое-либо произвольное значение (например, MyData). Таким образом, для других компонент, имеющих свойство DatabaseName можно будет указывать MyData в качестве значения этого свойства.

В то же время, если бы у нас не был определен псевдоним DATA1, или же по каким-либо причинам нам не хотелось бы его использовать, то мы могли бы «с нуля» создать псевдоним для данного приложения. Для этого потребуется указать те же параметры, что и при создании псевдонима средствами BDE Administrator, а именно тип драйвера, путь к БД и собственно псевдоним. Код получится примерно таким:

```
Database1.DatabaseName:='MyData1'; // псевдоним
Database1.DriverName:='STANDARD'; // драйвер Paradox
Database1.Connected:=true; // активируем компонент
Database1.Directory:='C:\Data'; // устанавливаем путь к файлам данных
```

Вместе с тем такое свойство, как DatabaseName предпочтительно все же задавать не во время выполнения, а в режиме разработки через инспектор объекта. Это позволит ссылаться на БД из других компонент, указывая в их свойстве DatabaseName, так же уже на этапе визуального проектирования приложения.

Одним из таких компонент, имеющим свойство DatabaseName, как раз и является Table. Если Database представляет собой базу данных в целом, то Table – это представление единичной таблицы из БД. Следует сразу отметить, что Table может обращаться к таблицам БД как через посредство компонента Database, ссылаясь на определенное в нем свойство DatabaseName, так и напрямую, если в свойстве DatabaseName компонента Table указать один из определенных в BDE псевдонимов. Разумеется, компонент Table имеет и другие свойства – они приведены в таблице 18.3.

Таблица 18.3. Свойства компонента Table

Свойство	Тип	Описание
Active	Boolean	Определяет, должно ли быть установлено подключение к базе данных
CanModify	Boolean	Указывает, может ли приложение изменять содержимое таблицы Paradox или dBase
DatabaseName	String	Определяет имя БД, ассоциированное с данным компонентом
DataSource	TDataSource	Определяет имя объекта источника данных для использования данной таблицы в качестве подчиненной при связи типа главный-подчиненный
DefaultIndex	Boolean	Определяет, должно ли производиться упорядочивание записей таблицы по первичному индексу

Exclusive	Boolean	Включает монопольный доступ к данной таблице Paradox или dBase
Exists	Boolean	Указывает, существует ли данная таблица в БД
FieldDefs	TFieldDefs	Указывает на список полей, определяющих данные
Filter	String	Определяет условие, по которому будет происходить выборка полей для показа
Filtered	Boolean	Определяет, является ли фильтр включенным
FilterOptions	TFilterOptions	Определяет набор флагов для фильтра
IndexDefs	TIndexDefs	Предоставляет информацию о индексах таблицы
IndexFieldCount	Integer	Указывает на количество полей, использованных в текущем индексе
IndexFieldNames	String	Определяет список полей, используемых в качестве индекса (через запятую)
IndexFields	array of Tfield	Список индексов таблицы
IndexFiles	TStrungs	Определяет список файлов с индексами для таблиц dBase
IndexName	String	Определяет вторичный индекс, по которому должно производиться упорядочивание (вместо первичного)
MasterFields	String	Определяет список полей (через запятую) в главной таблице, по которым должна устанавливаться связь типа главный-подчиненный
MasterSource	TDataSource	Определяет имя объекта источника данных для использования данной таблицы в качестве главной при связи типа главный-подчиненный
SessionName	String	Определяет имя сессии для данного компонента
ReadOnly	Boolean	Определяет режим доступа к таблице
TableName	String	Определяет имя таблицы (имя файла для таблиц dBase или Paradox)
TableType	TTableType	Определяет тип таблицы. Может принимать значения ttDefault (тип определяется по расширению файла), ttParadox, ttDBase, ttFoxPro и ttASCII

Здесь следует оговориться, что на самом деле некоторые свойства лишь унаследованы компонентом Table от своих предков. Соответственно, мы можем их встретить и в других компонентах, связанных с БД. В частности, это свойство Active, унаследованное от класса TDataSet – общего предка всех БД-компонент, предоставляющих непосредственных доступ к данным. От этого же предка происходит и свойство FieldDefs, позволяющее самостоятельно настроить список обрабатываемых полей. А группа свойств, связанных с отбором данных (Filter, Filtered и FilterOptions) являются частью класса TBDEDataSet – наследника класса TDataSet, ориентированного на работу с базами данных посредством драйверов BDE.

Остановимся на свойстве FilterOptions, которое содержит 2 флага:

- foCaseInsensitive – строки будут сравниваться без учета регистра символов;
- foNoPartialCompare – символ «звездочка» (*) в поле фильтра будет интерпретироваться именно как символ, а не как шаблон подстановки.

Свойства фильтров используются только при работе с BDE, поскольку в SQL-ориентированных компонентах подразумевается, что отбор данных будет производиться при помощи запросов SQL, что, в общем-то, вполне естественно, учитывая предназначение и возможности самого SQL.

ПРИМЕЧАНИЕ

> Еще одно важное свойство – FieldDefs, являющееся списком объектов – полей TField, будет рассмотрено вместе с самими полями позже, в главе, посвященной непосредственной работе с данными.

Для использования таблицы достаточно указать значения для свойств DatabaseName и TableName. Например, если продолжить пример с созданием псевдонима для компонента Database, то после последней строчки кода, устанавливающей значение для свойства Directory, достаточно добавить:

```
Table1.DatabaseName:='MyData1'; // устанавливаем значение БД на псевдоним
Table1.TableName:='Customer'; // в имени таблицы расширение не обязательно
Table1.Active:=true; // делаем таблицу подключенной
```

Дальнейшее использование компонента Table чаще всего сводится к тому, что он, представляя собой таблицу БД, позволяет производить над ней ряд манипуляций, включая редактирование данных, перемещение по записям и т.д. Для этого используются методы, которых у компонента Table, с учетом всех наследований, имеется около сотни. Впрочем, многие из них нельзя назвать повседневно необходимыми. Из наиболее востребованных можно отметить методы First и Last, Next и Prior, используемые для навигации по записям таблицы, а так же Append, Delete и Insert, которые используются для добавления и удаления записей. Следует отметить, что все эти методы являются унаследованными от класса TDataSet, что говорит о том, что они имеются и у множества других предназначенных для работы с БД компонент.

Альтернативы BDE

Как уже было сказано, Borland не считает BDE вполне современным и прогрессивным механизмом работы с БД. В качестве непосредственной замены BDE предлагается dbExpress – совокупность драйверов и компонентов, работающих с соединениями, транзакциями и запросами. С СУБД dbExpress общается посредством драйверов, которые для получения данных используют SQL. При этом на стороне клиентского приложения данные не кэшируются, что означает использование однонаправленных курсоров и невозможность непосредственной правки таблиц. Впрочем, технология dbExpress в любом случае не предназначена для работы с локальными БД. Среди поддерживаемых dbExpress СУБД можно отметить DB2, Oracle, MS SQL и MySQL. Разумеется, имеется поддержка и фирменной СУБД Interbase.

Впрочем, использование dbExpress для Interbase – не самое лучшее решение: дело в том, что в Delphi имеется еще одна технология, вернее – набор компонент, реализующих непосредственное взаимодействие с СУБД Interbase – IB Express. На палитре компонентов они находятся на закладке InterBase. Эти компоненты реализуют все возможности, имеющиеся у BDE, а так же позволяют использовать специфические для СУБД Interbase возможности, как-то хранимые процедуры и т.д. Кроме того, имеется набор компонентов InterBase Admin, при помощи которых можно производить манипуляции над самой СУБД Interbase.

Наконец, в Delphi предусмотрен еще одни основной механизм доступа к данным, а именно – ADO, компоненты которого расположены на одноименной закладке палитры компонентов. В общем и целом ADO можно рассматривать как вариант BDE в исполнении Microsoft. Правда, ADO общается с БД через интерфейс COM, что, возможно, не так оптимально, как работа напрямую из BDE (для случая с поддерживаемыми BDE СУБД), но при этом в качестве преимущества мы имеем то, что COM уже присутствует на любом Windows-ПК, в то время как BDE необходимо устанавливать отдельно. Собственно говоря, это и есть преимущество ADO над BDE, подобно всем остальным случаям с приложениями Microsoft, входящим в состав в Windows.

В любом случае, охватить все возможные пути работы с базами данных, предусмотренные в Delphi, в рамках настоящего издания не представляется возможным, особенно если учитывать тот факт, что помимо названных технологий, входящих в состав Delphi, имеются еще и альтернативные разработки. Поэтому в дальнейшем мы сосредоточим свое внимание на BDE, как на наиболее универсальном и широко распространенном варианте. Вместе с тем, мы рассмотрим в общих чертах работу с БД при помощи запросов на SQL, поскольку подобный подход приемлем для любых ныне встречающихся технологий доступа к данным, включая BDE, dbExpress, IB Express и ADO.

Компоненты доступа и представления данных

До настоящего момента мы говорили лишь о том, как получить некий абстрактный доступ к данным из приложения, опуская самую главную, с точки зрения конечного потребителя приложения БД, возможность – собственно представление данных в приложении. Для этих целей в VCL предусмотрено 2 группы компонентов – Data Access и Data Controls.

Доступ к данным

Для доступа к данным, представленным при помощи различных компонент – будь то BDE-ориентированные источники (например, Table), или ADO, IB Express, или dbExpress, используется один и тот же набор компонентов, расположенных на закладке Data Access:

- DataSource – источник данных;
- ClientDataSet – клиентский набор данных;
- DataSetProvider – провайдер набора данных;
- XMLTransform – преобразователь данных, представленных в виде XML в обычный пакет данных и обратно;
- XMLTransformProvider – провайдер данных для XML-документов, осуществляющий так же их обновление;
- XMLTransformClient – адаптер между XML-документом и провайдером.

Из этого списка нам интересно только первые 3 компонента, а именно DataSource, ClientDataSet и DataSetProvider. Используя набор из этих компонент, можно обеспечить доступ к данным. Причем в случае, когда речь идет о BDE и таблицах Paradox, как правило, достаточно использовать лишь один из перечисленных компонентов –

DataSource. Этот компонент имеет всего 4 собственных свойства – AutoEdit, DataSet, Enabled и State. Свойство Enabled похоже на свойство Active таблицы или Connected у базы данных, т.е. делает активным или неактивным соединение. А свойство AutoEdit, будучи включенным, обеспечивает возможность правки записей без написания какого-либо дополнительного кода. Свойство State информирует о том, в каком состоянии в текущий момент находится источник данных. Ну а самое важное свойство этого компонента – это, конечно же, DataSet, которое и определяет источник данных – таблицу, запрос и т.д.

Оставшиеся ClientDataSet и DataSetProvider могут понадобиться в том случае, если требуется обеспечить кэширование записей, например, для того, чтобы представить в виде таблицы источник данных типа dbExpress.

Рассмотрим пример, когда нам требуется обеспечить доступ к данным простой таблицы Paradox. Для этого нам на форме приложения понадобятся следующие компоненты: Database и Table из BDE, а так же DataSource из Data Access.

ПРИМЕЧАНИЕ

> Хотя для таблицы (компонента Table) можно указать один из определенных в BDE псевдонимов без помощи компонента Database, по сложившейся традиции, а так же в целях удобства управления приложением, все-таки предпочтительнее использовать связку из Database и Table.

Теперь для свойства AliasName компонента Database выберем название имеющегося у нас псевдонима БД (DATA1), а в качестве значения свойства DatabaseName так же напишем DATA1. Таким образом, компонент Database будет видеть «настоящий» псевдоним DATA1, а все остальные компоненты приложения – псевдоним DATA1, определенный посредством компонента Database.

Если такое положение вещей вас смущает, то в качестве значения DatabaseName можно указать любое произвольное значение, например, MyDatabase – в таком случае у других компонентов, имеющих свойство DatabaseName среди возможных значений этого свойства, будет значиться и MyDatabase.

Тем не менее, мы остановимся на начальном варианте, и перейдем к компоненту Table, для которого нам так же придется установить значения для 2 свойств – DatabaseName и TableName. Для первого укажем DATA1, для второго – customer. Чтобы убедиться, что все сделано правильно, попробуем активировать связь с БД, для чего установим в истину свойство Active. Если все было сделано верно (включая создание псевдонима DATA1 и таблицы customer, о чем речь шла в предыдущей главе), то не только свойство Active таблицы изменится на истину, но и свойство Connected компонента Database так же изменится на истину.

Последнее, что осталось сделать – это поместить на форму компонент DataSource и установить значение его свойства DataSet в Table1. Таким образом, мы получим действующую связку из 3 компонентов, обеспечивающих все этапы взаимодействия с БД – от организации локального псевдонима и управления им (Database), до выбора конкретной таблицы с возможностью управления ее параметрами (Table) и предоставления ее данных любым другим компонентам (DataSource).

Таблица DB Grid

Теперь настало время рассмотреть собственно компоненты, которым могут понадобиться данные для представления. Все они расположены на закладке Data Controls. Прежде всего, это, конечно, специальная таблица для баз данных – DBGrid. Этот компонент является дальнейшим развитием обычной таблицы (StringGrid), но предназначен исключительно для отображения и редактирования связанной с БД информации. Соответственно, у DBGrid нет таких свойств, как Cells, Cols и Rows, поскольку все, что выводит этот компонент – есть прямое отражение текущего содержимого связанной с ним таблицы БД.

В то же время, у компонента DBGrid предусмотрен целый ряд специальных свойств, предназначенных для взаимодействия с БД. Прежде всего, это свойство DataSource, в котором указывают имя компонента-источника данных. Так, если на форму, где уже имеются настроенные соответствующим образом невизуальные компоненты Database, Table и DataSource поместить таблицу, в свойстве DataSource которой указать DataSource1, то мы сразу же увидим содержимое таблицы customer (рис. 19.1).

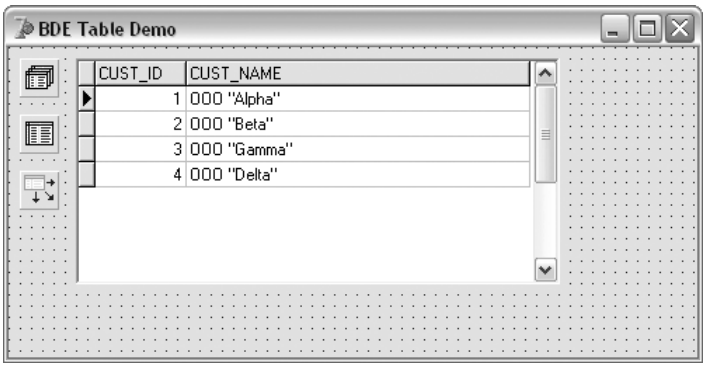

Рис. 19.1. Форма с таблицей БД

Следует сразу же отметить, что в качестве заголовков столбцов были использованы названия полей таблицы БД. Кроме того, можно увидеть, что столбец, содержащий числовые данные, имеет выравнивание по правому краю, а строковые – по левому. Таким образом, очевидно, что компонент DBGrid имеет более широкие возможности по оформлению таблиц, чем обычная таблица StringGrid. Возможно это благодаря другому свойству DBGrid – Columns, которое определяет оформление, количество и порядок следования столбцов с данными. Это свойство представляет собой коллекцию, состоящую из отдельных колонок таблицы. По умолчанию используется автоматический режим вывода, когда выводятся все поля данных с размерами, основанными на параметрах самих полей, заданных в выводимой таблице БД. Но поскольку во многих случаях выводить все поля не требуется, или же необходимо изменить параметры их вывода, как-то порядок следования, цвет, шрифт, или ширину поля, то все эти настройки доступны именно через свойство Columns. При этом каждый элемент этого свойства представляет собой объект типа TColumn, имеющий, в свою очередь, такие свойства, как заголовок (вместо стандартного названия поля в БД) выравнивание, цвет фона, шрифт, возможность правки и т.д. Доступ ко всем этим параметрам возможен через специальный редактор коллекций, который можно вызвать, дважды щелкнув по самой таблице в режиме разработки (рис. 19.2).

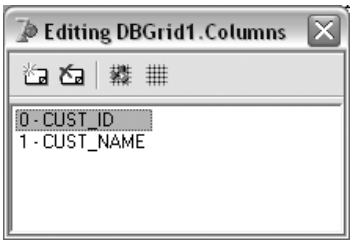

Рис. 19.2. Редактирование списка столбцов

На панели инструментов редактора столбцов имеется 4 кнопки – для добавления и удаления столбцов, а так же для автоматического заполнения и для сброса к начальным установкам. Выбирая поле из списка, и изменяя его свойства в инспекторе объекта, вы тем самым изменяете параметры отображения соответствующего столбца. А меняя в редакторе строки местам (путем перетаскивания мышкой), вы меняете порядок вывода полей в самой таблице.

Таким образом, мы рассмотрели 2 наиболее важных свойства компонента DBGrid. Что касается всех собственных свойств, имеющихся у таблицы для баз данных, то они перечислены в таблице 19.1.

Таблица 19.1. Собственные свойства компонента DBGrid

Свойство	Тип	Описание
Columns	TDBGridColumns	Задает параметры вывода столбцов с данными
DataSource	TDataSource	Определяет источник данных для отображения в таблице
FieldCount	Integer	Указывает на число столбцов с данными, выводимых в таблице
Fields	array of TField	Предоставляет доступ к информации ячейки, находящейся в указанном столбце
Options	TDBGridOption	Определяет различные параметры отображения и поведения таблицы
ReadOnly	Boolean	Определяет, будет ли у пользователя возможность править данные в таблице
SelectedField	TField	Предоставляет доступ к информации в выделенной ячейке
SelectedIndex	Integer	Определяет номер текущего столбца
TitleFont	TFont	Определяет шрифт, используемый для вывода заголовков столбцов таблицы

Здесь следует отдельно выделить свойство Options, позволяющее настроить целый ряд различных параметров. Оно имеет следующие флаги:

- dgEditing – Делает возможной правку данных прямо в таблице. Этот флаг игнорируется, если включен флаг dgRowSelect;
- dgAlwaysShowEditor – Таблица будет постоянно находиться в режиме редактирования. В противном случае пользователь должен будет нажимать F2, Enter, или щелкать мышкой по полю, чтобы ввести новое значение;
- dgTitles – Делает видимыми заголовки столбцов;

- dgIndicator – Добавляет колонку, в которой будет отображаться индикатор выбранной записи;
- dgColumnResize – Делает возможным изменение размеров столбцов пользователем;
- dgColLines – Столбцы будут отделены разделительными линиями;
- dgRowLines – Записи будут отделены разделительными линиями;
- dgTabs – Делает возможной навигацию по ячейкам при помощи клавиш Tab и Shift+Tab;
- dgRowSelect – Записи будут выделяться целиком. При этом правка данных в таблице становится невозможной (т.е. флаги dgEditing и dgAlwaysShowEditor будут проигнорированы);
- dgAlwaysShowSelection – Выбранная ячейка будет выделена цветом даже если фокус вводе не находится на таблице;
- dgConfirmDelete – Будет выдаваться предупреждение, если пользователь захочет удалить запись в таблице (при помощи Ctrl+Delete);
- dgCancelOnExit – Предотвращает запись пустых записей;
- dgMultiSelect – Делает возможным выбирать несколько записей одновременно (с использованием клавиши Ctrl).

Например, если таблица должна будет использоваться лишь для навигации по БД и просмотра значений, то будет рациональным установить флаг dgRowSelect. С одной стороны, это автоматически отключит возможность правки и ввода данных непосредственно в самой таблице пользователем, а с другой – будет визуально выделять текущую запись, при этом достаточно наглядно показывая пользователю, что правка невозможна.

В типичном случае все производимые при разработке приложения настройки компонента DBGrid сводятся к тому, что, поместив его на форму, указывают связанный источник данных, после чего при помощи редактора определяют состав и вид столбцов данных. В случае при необходимости так же выставляют нужные значения для флагов в свойстве Options.

Навигация по таблице данных

Хотя в ряде случаев для обеспечения возможности навигации по таблице достаточно использовать лишь стандартные средства, которыми располагает компонент DBGrid, в ряде случаев бывает полезным предоставить пользователю более наглядный элемент управления для навигации по данным и для их правки. Более того, подобный компонент будет просто необходим, если для предоставления данных используется не таблица, а набор отдельных элементов, отображающих данные из одного конкретного поля.

Для этих целей предусмотрен специальный компонент – DBNavigator. С его помощью можно перемещаться по записям таблицы а так же выполнять операции типа

вставки новой записи или подтверждения изменений. Внешне он представляет собой панель со следующими 10 кнопками:

- First – переход на первую запись в таблице;
- Prior – переход на предыдущую запись;
- Next – переход на следующую запись;
- Last – переход на последнюю запись;
- Insert – вставка новой записи перед текущей;
- Delete – удаление текущей записи с переходом на следующую;
- Edit – переводит источник данных в режим редактирования записи;
- Post – запись измененных данных из текущей записи в БД;
- Cancel – отмена изменений данных в текущей записи;
- Refresh – обновление данных в буфере источника.

Часть этих кнопок можно отключить, воспользовавшись свойством VisibleButtons. Еще одно свойство, влияющее на внешний вид компонента DBNavigator – это Flat. Установив его в истину, можно придать панели «плоский» вид. А при помощи свойства Hints можно задать пояснительный текст всплывающей подсказки для каждой кнопки.

Еще одно свойство – ConfirmDelete определяет поведение этого компонента: если для него установлено значение истины, то при попытке удаления записи (т.е. при нажатии на кнопку Delete) будет выдаваться соответствующее предупреждение.

Наконец, свойство DataSource, как и у всех других компонент представления данных БД, указывает на источник данных, связанных с данным компонентом.

Для примера добавим компонент навигации на форму с таблицей, подобной той, что изображена на рис. 19.1. Достаточно установить свойство DataSource компонента DBNavigator в то же значение, что и у одноименного свойства компонента DBGrid, чтобы получить связанно работающие компоненты. Например, при редактировании записей в таблице, состояние кнопок в навигационной панели будет изменяться в соответствии с возможными действиями (рис. 19.3).

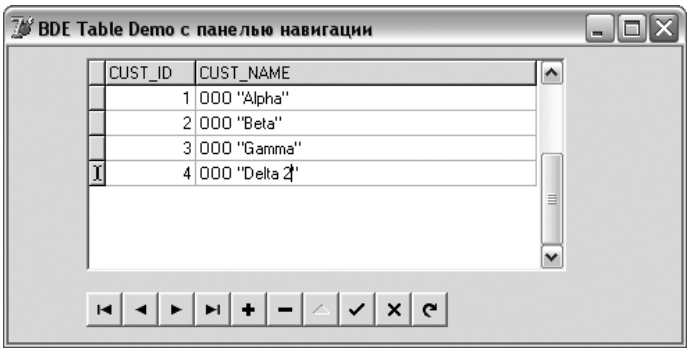

Рис. 19.3. Таблица DBGrid и панель навигации

Пример можно найти в каталоге *Demo\Part4\SimpleTable*.

Представление отдельных полей данных

Как мы уже знаем, для отображения таблиц БД в целом, используется табличный же компонент – DBGrid. В том же случае, когда надо отобразить содержимое лишь отдельных полей данных, используют соответствующие компоненты – DBEdit, DBImage, DBCheckBox и т.д., в зависимости от типа данных, которые требуется отобразить в том или ином случае.

Все эти компоненты являются специализированными вариантами обычных компонент, не связанных с БД:

- DBText – аналог текстовой подписи Label;
- DBEdit – аналог однострочного редактора Edit;
- DBMemo – аналог многострочного редактора (блокнота) Memo;
- DBImage – аналог компонента для вывода изображений Image;
- DBListBox – аналог списка ListBox;
- DBComboBox – аналог раскрывающегося списка ComboBox;
- DBCheckBox – аналог переключателя CheckBox;
- DBRadioGroup – аналог группы исключающих переключателей RadioGroup;
- DBRichEdit – аналог редактора форматированного текста RichEdit.

Основным отличием ориентированных на применение совместно с базами данных компонент является наличие у них свойства DataSource, при помощи которого они связываются с источником данных. Еще одно свойство – DataField, как раз и указывает на то поле, которое должно отображаться в данном компоненте. В остальном по своим функциональным способностям они повторяют свои не ориентированные на БД аналоги.

Что касается вариантов использования, то, например, мы можем заменить таблицу в предыдущем примере на пару компонент – DBText для отображения номера и DBEdit для вывода и редактирования названия. Соответственно, мы можем заменить табличное представление данных представлением типа «форма».

Для этого, удалив таблицу и разместив на форме компоненты DBText и DBEdit, установим для них обоих свойство DataSource в значение DataSource1, после чего для свойства DataField у метки выберем значение CUST_ID, а для этого же свойства у редактора – CUST_NAME. Не помешает также разместить на форме еще и 2 обычные метки (Label), при помощи которых можно вывести текст, поясняющий, что за информация выводится в том или ином поле (рис. 19.4).

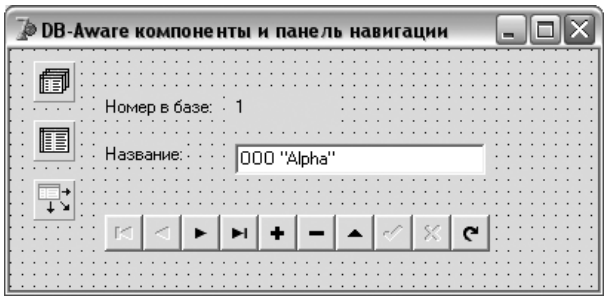

Рис. 19.4. Использование отличного от таблицы формата отображения данных

ПРИМЕЧАНИЕ

Здесь следует отметить тот факт, что данные могут автоматически преобразовываться в тот формат, который доступен компоненту. Например, целочисленное значение номера клиента было автоматически преобразовано в строку для вывода в DBText. Вместе с тем, возможности преобразований небезграничны, поэтому следует внимательно подбирать подходящие компоненты для отображения информации в каждом конкретном случае.

Компонент DBCtrlGrid

Особняком от других компонентов для отображения информации баз данных стоит компонент DBCtrlGrid, не имеющий прямых аналогов среди «обычных» компонентов VCL. Он состоит из набора однотипных панелей и позволяет отображать данные в произвольной форме. При этом каждая панель является платформой, на которой размещены простые БД-компоненты – такие, как DBText, DBEdit, DBCheckBox и т.п. У всех этих компонентов будет общий источник данных, который задается централизованно в свойствах самого DBCtrlGrid. Соответственно, останется лишь выбрать поля, которые помещенные на DBCtrlGrid компоненты будут отображать.

Во время разработки определяют единственную панель, по образу которой будут созданы ее точные копии. Например, можно взять за основу форму вывода таблицы клиентов в виде DBText и DBEdit, поместив на нее DBCtrlGrid, на котором, в свою очередь, расположить метку с редактором. В результате после запуска приложения эти 2 компонента будут продублированы на всех видимых панелях сетки (рис. 19.5).

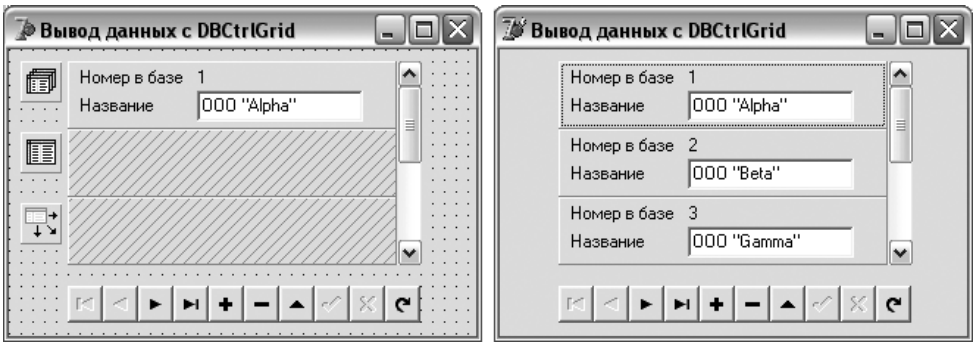

Рис. 19.5. Компонент DBCtrlGrid во время разработки (слева) и во время выполнения

Данные в таких панелях-ячейках могут располагаться как вертикально, так и горизонтально, либо вообще в качестве таблицы – в несколько рядов и колонок. Формат вывода панелей, как и их размеры, а так же ряд иных параметров этого компонента, определяется при помощи ряда свойств, перечисленных в таблице 19.2.

Таблица 19.2. Свойства компонента DBCtrlGrid

Свойство	Тип	Описание
AllowDelete	Boolean	Определяет, может ли пользователь удалить текущую запись, нажав Ctrl+Delete.
AllowInsert	Boolean	Определяет, может ли пользователь вставить новую запись, нажав клавишу Insert или добавить запись в конец, нажав Ctrl+Insert
ColCount	Integer	Определяет число столбцов с панелями
DataSource	TDataSource	Определяет источник данных для отображения
Orientation	TDBCtrlGridOrientation	Определяет порядок следования записей. Допустимые значения: goVertical (с вертикальной прокруткой) и goHorizontal (с горизонтальной прокруткой)
PanelBorder	TDBCtrlGridBorder	Определяет, должна ли быть рамка вокруг каждой панели. Допустимые значения: gbNone, gbRaised
PanelCount	Integer	Указывает на число видимых панелей
PanelHeight	Integer	Определяет высоту каждой панели в пикселях
PanelIndex	Integer	Определяет порядковый номе выбранной панели
PanelWidth	Integer	Определяет ширину каждой панели в пикселях
RowCount	Integer	Определяет число строк с панелями
SelectedColor	TColor	Определяет цвет фона активной панели
ShowFocus	Boolean	Определяет, должна ли отображаться дополнительная рамка вокруг панели при получении фокуса ввода

Размеры панелей определяются при помощи свойств PanelHeight и PanelWidth, а размеры компонента в целом определяются так же количеством панелей по горизонтали и по вертикали, определяемые через свойства ColCount и RowRount.

ПРИМЕЧАНИЕ

> Не рекомендуется помещать на панели ресурсоемкие компоненты вроде блокнота или изображения, особенно если панелей много. Так же следует учитывать, что все компоненты будут иметь один и тот же источник данных, определенный для самого DBCtrlGrid.

Пример с использованием данного компонента совместно с панелью навигации можно найти в каталоге *Demo\Part4\CtrlGrid*.

Связывание данных в таблицах

Важной особенностью работы с БД является связывание данных в таблицах. И хотя такое связывание осуществляется невизуальным компонентом Table, наглядно продемонстрировать связывание можно лишь с использованием компонентов, отобра-

жающих данные таблиц, т.е. с использованием, например, рассмотренного в этой главе компонента DBGrid.

Очевидно, что для связывания данных между таблицами нам нужно иметь более одной таблицы. Поэтому для начала откроем Database Desktop и создадим новую таблицу, предназначенную для связывания с уже имеющейся у нас таблицей клиентов.

Пусть это будет таблица счетов (Bill) для этих самых клиентов. Таким образом, в новой таблице нам понадобится, как минимум, 3 поля: номер счета (BILL_ID), идентификатор клиента (BILL_CUST) и сумма (BILL_SUMM). При этом поле номера счета будет иметь автоинкрементный тип, и являться индексом, а поле с номером клиента – целочисленный тип и так же быть индексированным. Наконец, поле суммы будет денежного типа.

После создания структуры таблицы останется заполнить ее произвольными значениями – достаточно будет ввести по 1-2-3 записи для каждого из клиентов (рис. 19.6). Достаточно лишь помнить, что число, вводимое в поле идентификатора клиента должно соответствовать одному из значений в поле CUST_ID таблицы клиентов, т.к. если за этим не следить, то получатся «потерянные» данные, не связанные с другими, что является нарушением целостности БД.

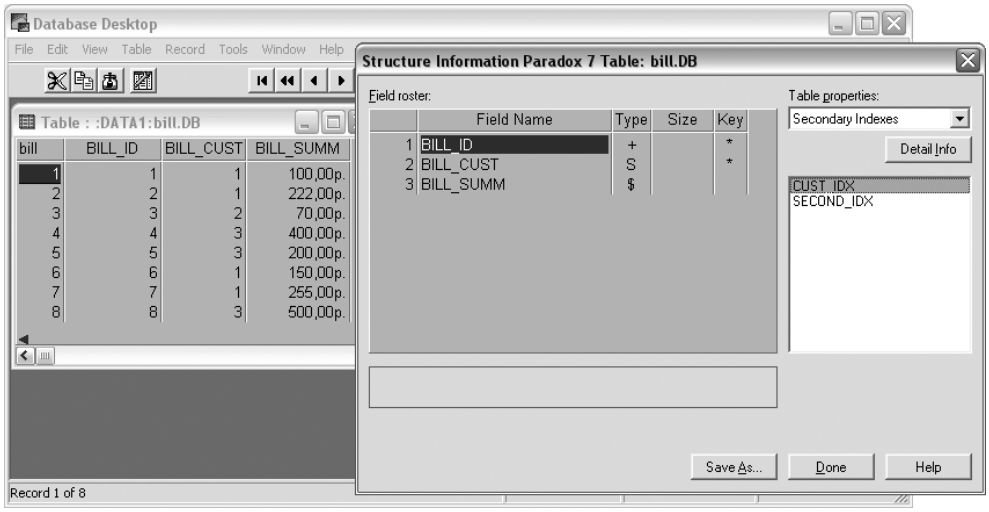

Рис. 19.6. Структура и содержимое таблицы счетов Bill

После подготовки исходных данных, перейдем к разработке приложения. Нам потребуется 1 компонент Database и по 2 компонента Table, DataSource и DBGrid. Для компонента Database установим те же значения, что и в предыдущем примере, т.е. AliasName и DatabaseName установим в DATA1, после чего свойство Connected можно установить в истину. Затем для первого «комплекта» из Table, DataSource и DBGrid так же устанавливаем значения, аналогичные предыдущему примеру, т.е. для Table1 установим свойство DatabaseName – в DATA1, а TableName – в customer.DB. Для DataSource1 установим свойство DataSet в Table1, и, наконец, для DBGrid1 установим свойство DataSource в DataSource1.

Затем подобную операцию следует провести для 2-го набора компонент, установив свойства подобным образом, за исключением того, что свойство TableName у компонента Table2 будет bill.DB, а во всех остальных случаях, разумеется, будут использованы имена этого набора (т.е. Table2, DataSource2, и т.д.). Если теперь для компонентов Table1 и Table2 установить значение свойства Active в истину, то в DBGrid1 и DBGrid2 мы увидим содержимое таблиц Customer и Bill.

Теперь займемся собственно связыванием данных. Для этого нам надо в подчиненной таблице, которую здесь представляет компонент Table2, указать на главную таблицу и сослаться на связанное поле, по которому и будет производиться отбор. Поскольку главная таблица предоставлена своим источником данных (DataSource1), то его и следует указать в свойстве MasterSource компонента Table2. Затем для свойства MasterFields установим значение CUST_ID. Наконец, в свойстве INDEX_NAME укажем вторичный индекс таблицы Bill, т.е. тот, который индексирует поле BILL_CUST. В нашем случае он называется CUST_IDX.

Таким образом, остается запустить приложение, чтобы убедиться, что связи работают. Если все сделано правильно, то при выборе записи в таблице с клиентами, в таблице счетов будут отображаться только те данные, которые относятся к этому клиенту. Пример можно найти в каталоге *Demo\Part4\OneToAny*, а с внешним видом этого приложения мы уже знакомы (см. рис. 17.2).

Компоненты синхронного просмотра

Специально для отображения связанной информации в БД, имеются 2 компонента, предназначенных именно для этих целей. Это компоненты DBLookupComboBox и DBLookupListBox. Оба они, хотя визуально и похожи на комбинированный и обычный списки, на самом деле, не являются потомками ни стандартных, ни БД-ориентированных компонентов, а происходят от общего для них класса TDBLookupControl, инкапсулирующего как список значений для просмотра, так и его механизм.

Соответственно, свойства этого класса наследуются обоими компонентами синхронного просмотра – как DBLookupComboBox, так и DBLookupListBox. Все они приведены в таблице 19.3.

Таблица 19.3. Общие свойства DBLookupComboBox и DBLookupListBox

Свойство	Тип	Описание
DataField	String	Определяет поле, которое будет отображаться данным компонентом
DataSource	TDataSource	Определяет источник данных для отображения в списке
Field	TField	Указывает на объект типа TField, который представляет собой данный компонент
KeyField	String	Определяет ключевое поле таблицы синхронного просмотра (ListSource), которое должно совпадать со значением поля, указанного в DataField
KeyValue	Variant	Представляет собой текущее значение ключевого поля
ListField	String	Определяет поле или список полей синхронного просмотра в таблице синхронного просмотра
ListFieldIndex	Integer	Определяет номер основного поля синхронного просмотра для случая, если в ListField указан список полей

ListSource	TDataSource	Определяет компонент источника данных (DataSource), связанный с таблицей синхронного просмотра
NullValueKey	TShortCut	Определяет сочетание горячих клавиш для сброса значения на неопределенное
ReadOnly	Boolean	Определяет, может ли пользователь изменять значения

Фактически, здесь следует запомнить лишь следующее: то, что отображается в самом компоненте синхронного просмотра (в обычном или в ниспадающем списке), задается парой значений для ListSource и ListField, а то, на основании чего происходит выборка текущего значения – в DataSource и DataField. При этом для связывания значений используется KeyField. Рассмотрим это на примере наших таблиц счетов и клиентов.

В предыдущем случае мы уже связывали таблицы средствами компонента Table. При этом первичным источником данных («мастером») являлась таблица клиентов, выбирая запись в которой, мы могли видеть соответствующие ей значения в подчиненной таблице счетов. Но теперь рассмотрим такой вариант, когда первичной информацией являются сами счета. В таком случае, в общем-то, достаточно одной таблицы со счетами. Единственная проблема состоит в том, что в ней виден лишь номер клиента, что удобно машине, но не годится для человека. Соответственно, название клиента можно отображать, воспользовавшись как раз компонентом DBLookupComboBox.

Прежде всего, нам понадобится привычный уже набор из 5 компонент данных, а именно Database и 2 пары Table и DataSource. Первую пару (Table1 и DataSource1) в данном случае мы настроим на таблицу счетов, а вторую – на клиентов. При этом никаких значений для MasterSource и MasterFields нам в данном случае устанавливать не потребуется. После этого пометим на форму таблицу DBGrid и свяжем ее с таблицей счетов, установив свойство DataSource в значение DataSource1. Если база данных подключена, а таблицы активны, то в таблице сразу же будет отображена вся информация о счетах.

Теперь перейдем к собственно синхронному просмотру. Для этого поместим на форму компонент DBLookupComboBox и установим для его свойств следующие значения: DataSource – DataSource1, DataField – BILL_CUST, ListSource – DataSource2, ListField – CUST_NAME, и, наконец, свойство KeyFiled установим в CUST_ID.

Остается лишь запустить приложение и убедиться, что при выборе той или иной записи в таблице счетов, в комбинированном списке синхронного просмотра отображается название соответствующего клиента (рис. 19.7).

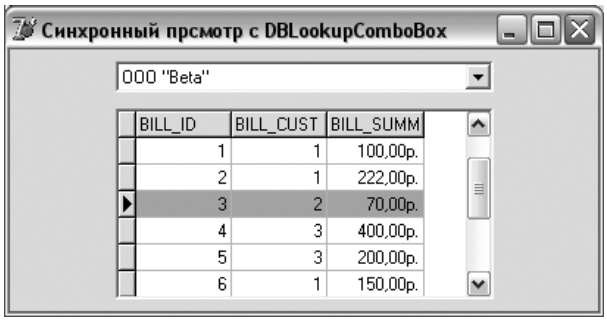

Рис. 19.7. Синхронный просмотр

Если же раскрыть сам список, то в нем можно будет увидеть весь список клиентов, а если выбрать из него другое значение, то цифра в колонке BILL_CUST так же поменяется. Таким образом, связь получается полноценная, в обе стороны.

Помимо рассмотренных свойств, отвечающих за организацию связи между данными, у компонента DBLookupComboBox имеются и собственные свойства, относящиеся к его визуальной части. Это DropDownAlign, DropDownRows и DropDownWidth, которые отвечают, соответственно, за выравнивание элементов в раскрывающемся списке, за их количество в нем и за ширину окна списка. Кроме них, для чтения во время выполнения программы доступны еще 2 свойства – ListVisible, указывающее на то, раскрыт ли список в данный момент, и Text, содержащее текущее значение списка в виде текстовой строки.

Что касается компонента DBLookupListBox, то с точки зрения организации синхронного просмотра он полностью аналогичен компоненту DBLookupComboBox. Различия касаются лишь визуальной формы представления, а так же собственных свойств, коих у синхронного списка всего 3 – BorderStyle, RowCount и SelecteItem. Первое отвечает за наличие обрамляющей рамки у списка, второе – указывает на количество видимых в списке рядов, а третье аналогично свойству Text у комбинированного списка, т.е. содержит выбранное значение в виде строки.

Модуль хранения компонентов данных

До настоящего момента мы рассматривали лишь простейшие случаи, с использованием 1-2 таблиц и такого же небольшого числа источников данных. Если же говорить о реальных приложениях БД, то число невизуальных компонентов, используемых в программе для доступа к данным, нередко исчисляется десятками. Такое их изобилие грозит превратить форму главного окна в одно сплошное нагромождение компонент, изрядно мешая работе. Кроме того, одни и те же компоненты могут понадобиться в разных окнах приложения, в то время, как включать в список используемых модулей главное (или любое другое) окно только лишь для ссылки на данные не представляется идеальным вариантом. Поэтому в Delphi предусмотрено специальная оконная форма – DataModule, предназначенная исключительно для размещения на ней невизуальных компонент для доступа к данным.

Отличие окна DataModule от обычной формы состоит в том, что на нем можно размещать только невизуальные компоненты. Это могут быть не только компоненты для доступа к данным, но, в принципе, и любые другие, необходимые в разных частях приложения.

На таком окне можно совершенно свободно, без лишней скученности и без оглядок на пользовательский интерфейс, расположить весьма внушительное число необходимых компонентов (рис. 19.8).

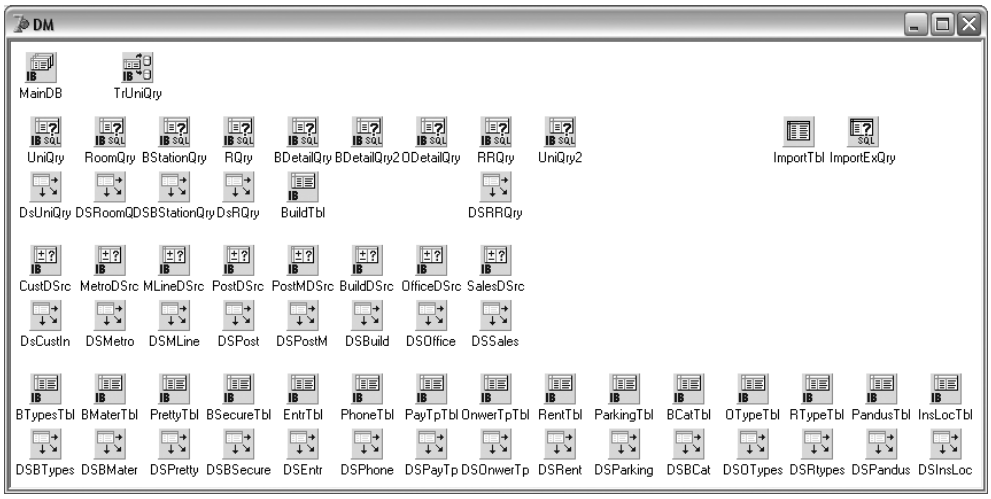

Рис. 19.8. DataModule может вместить в себя множество компонентов доступа к данным

Для создания окна DataModule следует из меню File → New выбрать пункт Data Module. После чего достаточно будет назначить имя этому модулю (например, DM) и сохранить файл, назвав его, скажем, data.pas. После этого, как и в случае с обычными формами, можно будет его включать в конструкцию uses. После этого становится возможным ссылаться на источники данных через стандартную точечную нотацию, используя имя модуля данных:

```
DBGrid1.DataSource:=DM.DataSource;
```

Пример, использующий вынесенные в отдельный модуль невизуальные компоненты, находится в каталоге *Demo\Part4\AppDM*.

Работа с данными

Теперь, когда нам известны все основные компоненты, необходимые для работы с СУБД – как обеспечивающие доступ к данным, так и их отображение, можно перейти к вопросам практического характера. Прежде всего, это вопросы, касающиеся полей данных, а так же состояния наборов данных, программная навигация, сортировка, поиск и фильтрация. Так же, разумеется, будут рассмотрены правка, добавление и удаление записей.

Состояния и режимы набора данных

Как мы уже знаем, одним из основных компонентов Delphi для доступа к данным является Table. Этот компонент происходит от общего для всех наборов данных класса – TDataSet – набор данных. Именно на уровне набора данных информация из БД представляются как совокупность строк и столбцов. В этом базовом классе так же сосредоточены все основные свойства и методы для работы с наборами данных, включая управление состоянием набора, поиск, фильтрацию, сортировку и изменение данных.

ВНИМАНИЕ

> Набор данных в Delphi – это некая логическая таблица, с которой может работать приложение. При этом порядок следования записей в наборе данных может отличаться от имеющегося в реальной таблице, поскольку зависит от того, используется ли сортировка. Равно как и количество записей в наборе зависит от наличия и параметров фильтрации.

Основными свойствами набора данных как такового являются Active и State, а применительно к записям – RecordCount и RecNo. В частности, свойство RecordCount указывает на текущее количество записей в наборе данных (которое может не совпадать с количеством записей в таблице БД благодаря возможной фильтрации), а свойство RecNo указывает на индекс активной записи.

Что касается свойства Active, то мы уже рассматривали его в контексте компонента Table и знаем, что оно отвечает за непосредственное подключение к БД. Значение этого свойства может быть установлено как на этапе разработки приложения, так и во время выполнения. При этом следует учитывать, что попытка установить свойство Active в истину может вызвать исключительную ситуацию, если не указана, как минимум, физическая таблица с данными (при помощи свойства TableName):

```
Table1.TableName='..\DB1\bill.db'; // указываем физическую таблицу БД
Table1.Active:=true; // делаем набор активным
```

Если же поддерживать активное соединение по ходу дальнейшего выполнения программы нет необходимости, то можно вернуть состояние в неактивное, присвоив свойству Active значение ложь. Эту операцию необходимо сделать и в том случае, если требуется изменить исходный источник с данными:

```
Table1.Active:=false; // отключаемся
Table1.TableName='..\DB1\customer.db'; // меняем таблицу БД
Table1.Active:=true; // включаем внось
```

Альтернативным методом изменения состояния набора данных является использование методов Open и Close:

```
Table1.Close; // отключаемся
Table1.TableName='..\DB1\customer.db'; // меняем таблицу БД
Table1.Open; // включаем внось
```

Фактически, эти методы выполняют ту же работу, что и изменение свойства Active, т.е. обращение к методу Open устанавливает свойство Active в истину, а Close – в ложь. При написании программного кода принято использовать именно эти методы, а не изменять значение свойства Active.

Другое свойство набора данных – State – показывает текущее состояние набора данных, или режим его работы. Оно имеет тип TDataSetState и доступно для чтения во время выполнения программы. Основными значениями State являются следующие:

- dsInactive – набор данных закрыт;
- dsBrowse – набор данных доступен для промотра, но не находится в состоянии изменения;
- dsEdit – текущая запись может быть изменена;
- dsInsert – добавлена, но еще не отправлена в таблицу новая запись;

- dsSetKey – осуществляется поиск записи (только для Table);
- dsCalcFields – осуществляется расчет полей;
- dsFilter – производится фильтрация записей;
- dsOpening – начат но еще не завершен процесс открытия набора данных.

При помощи свойства State во время выполнения программы можно получить текущее состояние набора данных, а само изменение состояние инициализирует событие OnStateChange, которое происходит для связанного с источником данных компонента DataSource.

Рассмотрим работу с источником данных на примере, для которого нам потребуются компоненты Table, DataSource, DBGrid и OpenDialog. При этом компоненты для баз данных должны быть связаны между собой, т.е. у DataSource1 в свойстве DataSet должно быть указано Table1, а у DBGrid1 свойство DataSource должно иметь значение DataSource1. Теперь разместим на форме кнопку (Button), которая будет вызывать диалог выбора файла и производить работу по подключению данных, для чего нам понадобится обработчик события OnClick. А для компонента DataSource1 мы сделает обработчик события OnStatusChange, который будет выводить текущее состояние набора данных в заголовок окна. Вариант кода приведен в листинге 20.1.

Листинг 20.1. Подключение источника данных и слежение за его состоянием

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 if not OpenDialog1.Execute then exit;
 Table1.Close;
 Table1.TableName:=OpenDialog1.FileName;
 Table1.Open;
end;
procedure TForm1.DataSource1StateChange(Sender: TObject);
begin
 case DataSource1.State of
  dsInactive: Caption:='Отключено';
  dsBrowse: Caption:='Просмотр данных';
  dsEdit: Caption:='Правка данных';
  dsInsert: Caption:='Вставка данных';
  dsOpening: Caption:='Открытие источника данных';
  else
  Caption:='Состояние неопределено';
 end;
end;
```

Пример программы находится в каталоге *Demo\Part4\DataSource*.

Поля и класс TField

Если для данных в целом используют компоненты типа Table, т.е. потомки TDataSet, то для, полей, которые представляют собой отдельные столбцы данных в таком на-

боре, определен класс TField. В свою очередь, от этого класса происходят типизированные классы полей типа TIntegerField, TStringField и т.п.

Для доступа к полям записей у набора данных имеется ряд специальных методов и свойств, доступных во время выполнения. Чаще всего используются метод FieldByName, позволяющий обратиться к полю по его имени. Альтернативным и зачастую менее надежным способом является использование массива Fields, обеспечивающего доступ к полю по его порядковому номеру. И в том и в другом случае мы получим объект типа TField. Основные свойства этого класса приведены в таблице 20.1.

Таблица 20.1. основные свойства TField

Свойство	Тип	Описание
Alignment	TAlignment	Определяет выравнивание при выводе значения поля в визуальном компоненте
AsBCD	TBcd	Содержит значение поля в двоичном виде (BCD)
AsBoolean	Boolean	Содержит значение поля в виде булева значения
AsCurrency	Currency	Содержит значение поля в виде Currency
AsDateTime	TDateTime	Содержит значение поля в виде даты и времени
AsFloat	Double	Содержит значение поля в виде вещественного числа
AsInteger	Integer	Содержит значение поля в виде целого
AsString	String	Содержит значение поля в виде строки
AsVariant	Variant	Содержит значение поля в виде вариантного типа
Calculated	Boolean	Определяет, является ли поле вычисляемым
CanModify	Boolean	Указывает, может или нет быть изменено значение в данном поле
DataSet	TDataSet	Определяет набор данных, которому принадлежит данное поле
DataType	TFieldType	Указывает на тип данных поля
DisplayLabel	String	Определяет текст, выводимый в качестве заголовка столбца в таблице DBGrid
DisplayWidth	Integer	Определяет число символов, необходимое для вывода значения поля в визуальном компоненте
FieldName	String	Определяет имя поля в физической таблице БД
Index	Integer	Определяет порядковый номер поля в наборе данных
IsIndexField	Boolean	Указывает, является ли данное поле индексированным
IsNull	Boolean	Возвращает истину, если текущее значение поля пустое
ReadOnly	Boolean	Определяет, может ли поле быть изменено пользователем
Value	Variant	Содержит текущее значение поля
Visible	Boolean	Определяет, должно ли это поле быть видимым при отображении в таблице DBGrid

Здесь, прежде всего, следует выделить группу из 8 свойств, начинающихся с As. Все они, по сути, являются аналогами функций приведения к типу, поскольку, объект TField универсален и может содержать данные любого типа из встречающихся в СУБД. Например, если речь идет о полях из нашей таблицы bills, то обратиться к

полям этой таблицы для получения их значений в «естественном» виде следует использовать подобный код:

```
x:=Table1.FieldByName('BILL_CUST').AsInteger;
y:=Table1.FieldByName('BILL_SUMM').AsCurrency;
```

В то же время, если значения этих полей понадобятся нам для вывода в строковом виде (например, в поле однострочного редактора или), то ничего не мешает нам сразу же использовать приведение к нужному типу. В данном случае (с редактором и меткой) – к строковому:

```
Label1.Caption:=Table1.FieldByName('BILL_CUST').AsString;
Edit1.Text:=Table1.FieldByName('BILL_SUMM').AsString;
```

Если же при работе с записью нам не требуется явно приводить тип, то вполне подойдет свойство Value, содержащее текущее значение поля в БД. Оно же является одним из наиболее часто используемых, благодаря универсальности этого типа данных. Например, мы можем проверить значение поля на соответствие какому-либо условию, просто написав:

```
if Fields[1].Value > 100 then Fields[1].Value:=200;
```

При этом, правда, следует не забывать про то, что не всякий тип может быть приведен к нужному (в данном случае – к целому или вещественному числу).

Свойства DataSet и FieldName отвечают за привязку объекта к конкретным данным, а Calculated и IsIndexFiels позволяют определить параметры поля в СУБД. Большинство остальных свойств так или иначе влияют на параметры вывода информации из набора данных в визуальные компоненты наподобие DBGrid.

Что касается методов, то тут следует отметить метод Clear, устанавливающий значение поля в Null, и, пожалуй, IsValidChar, при помощи которого можно проверить, является ли указанный в качестве аргумента символ допустимым для данного поля:

```
if Table1.FieldByName('BILL_SUMM').IsValidChar('1') then caption:='Yes!'
  else caption:='No.'
```

В данном случае заголовок окна получит надпись «Yes!», поскольку символ «1» можно использовать для ввода в поле числового типа. Но если бы мы проверяли на допустимость ввода какой-либо буквы, то получили бы отрицательный ответ.

Типы полей и типы данных

Как уже было отмечено, тип Field является лишь предком для целого ряда (всего их около 30) типизированных полей. При этом каждый тип поля в Delphi соответствует определенному типу данных, используемому в той или иной СУБД. Чаще всего используются следующие классы:

- TBLOBField – поле BLOB-объекта;
- TMemoField – поле типа Memo;
- TGraphicField – графическое поле;
- TStringField – поле строкового значения;
- TBCDField – поле BCD-значения;
- TDateTimeField – поле даты и времени;

- TFloatField – поле вещественного числа;
- TCurrencyField – поле денежной суммы;
- TIntegerField – поле целого числа;
- TAutoIncField – поле автоинкрементного значения.

Не все перечисленные классы происходят напрямую от TField – для некоторых из них существует «промежуточный» предок. Например, для всех числовых типов определен общий класс TNumericField.

В то же время, имеющееся у класса TField свойство DataType, позволяет получить информацию о том, какого типа значение хранится в поле. Таким образом, обращаясь к методу базового класса, можно не задумываться над тем, с каким конкретным типом объекта мы имеем дело. В Delphi 7 оно определено следующим образом:

```
type TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger, ftWord,
ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime, ftBytes,
ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo, ftParadoxOle,
ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar, ftWideString, ftLargeint,
ftADT, ftArray, ftReference, ftDataSet, ftOraBlob, ftOraClob, ftVariant,
ftInterface, ftIDispatch, ftGuid, ftTimeStamp, ftFMTBcd);
```

Очевидно, что названия типов образуются от префикса ft и названия типа данных. Соответственно, если нам надо получить информацию о том, принадлежит ли интересующее нас поле к тому или иному типу, скажем, к Currency, то достаточно написать выражение вида:

```
if Table1.FieldByName('BILL_SUMM').DataType = ftCurrency then caption:='$';
```

Свойство DataType может пригодиться и в том случае, если нам требуется проверить, существует ли возможность привести данные поля к какому-то определенному виду. Например, если поле содержит целое или вещественное число, то оно может быть представлено как Currency, а если строкового, двоичного или какого-либо еще типа, то нет. Соответственно, проверка может выглядеть следующим образом:

```
var sum: Currency;
...
if Table1.FieldByName('BILL_SUMM').DataType in [ftInteger, ftWord, ftFloat,
ftCurrency] then sum:=Table1.FieldByName('BILL_SUMM').AsCurrency;
```

Определение типа поля актуально в том случае, если используются динамические поля, которые создаются автоматически при открытии набора данных. Однако поля можно определить жестко на этапе разработки приложения, в таком случае они будут статическими, и иметь какой-либо вполне определенный тип.

Для определения статических полей следует воспользоваться редактором полей, который можно вызвать двойным щелчком по компоненту Table. Внешне он напоминает редактор списка столбцов компонента DBGrid (см. рис. 19.2), что, впрочем, и не удивительно, так как в обоих случаях мы имеем дело с редактором коллекций. Выбрав их контекстного меню редактора полей пункт Add Fields, мы получим диалоговое окно со списком всех имеющихся в текущей таблице БД полей. Выбрав нужные поля, остается нажать OK и приступить к исследованию получившегося списка. В нем будут находиться выбранные поля (обозначенные по своим заголовкам в таблице БД), являющиеся объектами какого-либо из типов полей. Например, для таблицы клиентов это могут быть поля типа TAutoIncField (для CUST_ID) и TStringField (для

CUST_NAME). Если щелкнуть по названию поля в списке, то в инспекторе объектов мы увидим все его опубликованные свойства. При этом автоматически будут созданы и помещены в объявление класса формы соответствующие переменные:

```
type
  TForm1 = class(TForm)
    Table1: TTable;
    Table1CUST_ID: TAutoIncField;
    Table1CUST_NAME: TStringField;
    ...
  end;
```

Соответственно, в дальнейшем мы сможем оперировать именно этими переменными, а не обращаться к полям таблицы при помощи методов FieldByName и подобных способов:

```
Caption:=Table1CUST_NAME;
```

Кроме того, определяя статические поля на уровне набора данных, становится возможность определить ряд параметров отображения непосредственно на этом этапе. Иначе говоря, задать выравнивание можно для самого поля, а при необходимости, (например, при выводе в таблицу DBGrid) оно будет учтено. Здесь же можно задать порядок следования полей и ограничить их состав.

ПРИМЕЧАНИЕ

> Разумеется, можно определить параметры вывода и для динамических полей, но это менее удобно, поскольку на этапе разработки их свойства недоступны, в то время, как параметры статических полей можно определить визуально при помощи инспектора объектов.

Единственным недостатком статических полей по сравнению с динамическими является то, что в случае изменения структуры таблицы с данными, в программе может возникнуть исключительная ситуация.

Сортировка

Изначально порядок расположения записей в наборе данных бывает неопределенным. Так, для таблиц одних СУБД (например, dBASE) записи располагаются в порядке поступления в файл таблицы, а в других (например, в Paradox) они сортируются по первичному индексу. Однако очень часто требуется вывести записи в определенном порядке, для чего используется сортировка данных.

Сортировкой называют упорядочивание записей по определенному полю в порядке убывания или возрастания в нем значений. Возможна также сортировка по нескольким полям одновременно. В таком случае сначала сортируется первое поле, затем группы записей с одинаковыми значениями в первом поле сортируются по второму полю, и т.д.

Для набора данных типа Table сортировка выполняется автоматически по выбранному индексу. Соответственно, если изменить индекс, например, при помощи свойств IndexFieldNames или IndexName, то записи будут упорядочены заново. В качестве значения для IndexName указывают имя индекса, а для IndexFieldNames указывают имена полей, из которых индекс состоит. Здесь следует отметить, что поскольку в

таблицах Paradox первичный индекс не имеет имени, то для сортировки по нему можно использовать только свойство IndexFieldNames.

Если сортировку требуется провести по нескольким полям, то каждое из них должно быть включено в составной индекс. Из этого же следует, что при использовании компонента Table сортировка возможна только по индексированным полям.

ПРИМЕЧАНИЕ

> Помимо компонента Table, для наборов данных имеется SQL-ориентированный компонент Query. В случае использования Query сортировка производится при помощи языка SQL и может проводиться по любым полям. Вопросы использования компонента Query и языка запросов SQL будут рассмотрены в следующей главе.

Для примера возьмем таблицу счетов и попробуем отсортировать ее по всем возможным индексам. Для этого нам понадобятся компоненты Table, DataSource и DBGrid, а так же RadioGroup. Для Table установим значение свойства DatabaseName в DATA1, TableName – в bill.db, а Active – в истину. После этого свяжем DBGridc Table через компонент DataSource.

Как видно, изначально данные отображаются точно так же, как хранятся в таблице – последовательно с 1 по 8-я запись (на самом деле, в данном случае это заслуга первичного индекса, построенного по полю BILL_ID). Теперь попробуем произвести сортировку по вторичным индексам – CUST_IDX и SECOND_IDX, определенных в структуре таблицы БД, для чего определим 3 варианта в группе переключателей, назвав их «по умолчанию», «индекс 1» и «индекс 2». Теперь в процедуре для события OnClick группы напишем следующий код:

```
case RadioGroup1.ItemIndex of
  0: Table1.IndexName:='';
  1: Table1.IndexName:='CUST_IDX';
  2: Table1.IndexName:='SECOND_IDX';
end;
```

Если теперь запустить приложение и выбрать какой-либо вариант сортировки, то можно убедиться, что данные в таблице будут отсортированы по 1-му, 2-му, или 3-му столбцу, в зависимости от выбранного положения переключателя (рис. 20.1).

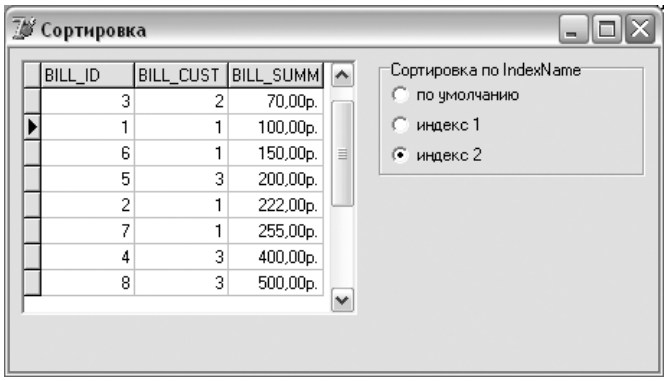

Рис. 20.1. Сортировка по сумме счета (индекс SECOND_IDX)

Недостатком использования свойства IndexName в данном случае состоит в том, что для таблиц Paradox, как и в данном случае, не удастся задействовать поле с первичным индексом. В таком случае можно воспользоваться свойством IndexFieldNames, для чего добавим еще одну группу, на этот раз – из 4 переключателей, а в обработчике для нее напишем следующий код:

```
case RadioGroup2.ItemIndex of
  0: Table1.IndexFieldNames:='';
  1: Table1.IndexFieldNames:='BILL_CUST';
  2: Table1.IndexFieldNames:='BILL_ID;BILL_CUST';
  3: Table1.IndexFieldNames:='BILL_SUMM;BILL_CUST';
end;
```

Теперь в 2 случаях (варианты 2 и 3) можно производить сортировку по нескольким полям сразу. Так же следует отметить, что свойства IndexName и IndexFieldNames являются взаимоисключающими, т.е. если установить какое-либо значение для одного из этих свойств, значение другого автоматически сбрасывается. Исходный код этой программы находится в каталоге *Demo\Part4\Sort*.

Что касается направления сортировки – по возрастанию или по убыванию, то за это отвечает флаг ixDescending свойства Options определения индекса. Эти определения, в свою очередь, задаются через свойство IndexDefs.

Навигация

Навигация по набору данных состоит в том, что выбирается указатель на ту или иную запись в таблице БД. Этот указатель так же называется курсором и определяет запись, с которой будут производиться операции в данный момент.

Мы уже рассматривали навигацию при помощи специального компонента – DBNavigator. Теперь рассмотрим, как можно осуществлять ее программным способом, обращаясь непосредственно к методам набора данных.

Для перемещения указателя текущей записи можно использовать следующие процедуры набора данных:

- First – устанавливает указатель на первую запись;
- Next – устанавливает указатель на запись, следующую за текущей;
- Last – устанавливает указатель на последнюю запись;
- Prior – устанавливает указатель на запись, предшествующую за текущей;

Для перемещения на несколько записей сразу используют функцию MoveBy – она принимает в качестве аргумента число, на которое должно произойти перемещение, а возвращает число записей, на которое реально переместился курсор (разница может быть вызвана, например тем, что конец набора данных был достигнут «досрочно»). Перемещение при помощи этой функции можно производить в обоих направлениях:

```
Table1.MoveBy(10); //перемещение указателя на 10 записей вперед
Table1.MoveBy(-3); //перемещение указателя на 3 записи назад
x:=Table1.MoveBy(y); //использование возвращаемого значения
if x<>y then Caption:='Перемещено только на '+IntToStr(x)+' записей';
```

При перемещении указателя учитывается текущее состояние набора данных, т.е. порядок сортировки и ограничения, наложенные фильтрами.

ВНИМАНИЕ

> Следует учитывать, что указатель привязывается именно к записи, а не к ее расположению в наборе данных. Соответственно, если после сортировки порядок записей изменится, указатель будет ссылаться на ту же запись, что и до сортировки. А вот после фильтрации указатель всегда указывает на первую запись в наборе данных.

Для примера возьмем все ту же таблицу счетов, задействовав компоненты Table, DataSource и DBGrid. Еще нам понадобятся 3 кнопки. Первая будет перемещать курсор на следующую запись, а вторая – на предыдущую. Соответственно, код для первой кнопки будет выглядеть так:

```
Table1.Next;
```

А у второй, соответственно:

```
Table1.Prior;
```

По сути, таким образом мы просто сделали свой вариант навигации без использования готового компонента DBNavigator. Однако программное управление курсором позволяет производить гораздо более сложные манипуляции. Например, при помощи последовательного перебора записей можно вычислить их суммарное значение. Поэтому третья кнопка будет использоваться как раз для того, чтобы подсчитать итоговую сумму всех счетов.

Для реализации этого замысла нам потребуется организовать цикл для обхода всех записей, начиная с первой, каждый раз добавляя значение поля BILL_SUMM к переменной. В завершение останется вывести результат в заголовок окна. В итоге мы получим процедуру, приведенную в листинге 20.2.

Листинг 20.2. Последовательный обход всех записей в источнике данных

```
procedure TForm1.Button3Click(Sender: TObject);
var
 i: integer;
 x: currency;
begin
 Table1.First; // начинаем с 1-й записи
 x:=0;
 for i:=0 to Table1.RecordCount-1 do begin
   x:=x+Table1.FieldByName('BILL_SUMM').AsCurrency;
   Table1.Next; // не забываем перевести курсор на следующую запись
 end;
 Caption:=CurrToStr(x);
end;
```

Для того чтобы пройтись по всем записям, здесь мы воспользовались таким свойством набора данных, как RecordCount. Это вполне логичный вариант, однако, он не совсем идеален с той точки зрения, что нам требуется дополнительная переменная. В то же время, у набора данных имеется такое свойство, как Eof, при помощи которого

можно узнать, достигнут или нет конец. Соответственно, чтобы избавиться от лишней переменной в своем коде, мы можем использовать цикл while:

```
while not Table1.Eof do begin
  x:=x+Table1.FieldByName('BILL_SUMM').AsCurrency;
  Table1.Next;
end;
```

В любом случае, по завершении цикла мы получим нужный нам результат в переменной x. Исходный код приведен в примере в каталоге *Demo/Part4/Navigate*.

Помимо свойства Eof, существует и свойство Bof, проверяющее, находится ли указатель в начале набора данных. Он может потребоваться, например, в том случае, если необходимо произвести подсчет в обратном порядке:

```
Table1.Last;
while not Table1.Bof do begin
  x:=x+Table1.FieldByName('BILL_SUMM').AsCurrency;
  Table1.Prior;
end;
```

Обратите внимание, что в данном случае курсор заблаговременно переводится на последнюю запись в наборе данных.

Фильтрация

Фильтрацией называют определение ограничений для записей, отбираемых для набора данных. По умолчанию фильтрация отключена, однако всегда можно ею воспользоваться, определив параметры фильтра в свойстве Filter и установив другое свойство – Filtered – в истину.

ПРИМЕЧАНИЕ

> При работе с запросами SQL посредством компонента Query или ему аналогичного, фильтрация набора данных действует поверх ограничений, заданных в самом SQL-запросе. Иначе говоря, если при помощи запроса уже был произведен «отсев» данных, то фильтрация дополнительно ограничит число выводимых записей.

В качестве выражения, определяющего фильтр, используется строковое значение, составленное по определенным правилам. В состав фильтра могут входить такие элементы, как имена полей таблиц, литералы, скобки, операции сравнения, а так же арифметические и логические операции. Под литералом здесь понимается заданное явно значение – число, строка или символ. Из арифметических операций доступны такие, как сложение, вычитание, умножение и деление. В качестве логических операций можно использовать AND, OR и NOT. Ну а скобки используются, как обычно, для изменения порядка выполнения операций.

Для примера рассмотрим несколько вариантов фильтров:

```
BILL_SUMM > 150
BILL_CUST > 2 AND BILL_CUST < 5
```

В первом случае будут отобраны только те ряды, у которых в поле BILL_SUMM значение превышает 150, а во втором – ряды, поле BILL_CUST которых имеет значения

больше 2 и меньше 5. Если в выражении фильтра используются строковые или символьные литералы, то они должны быть заключены в одинарные кавычки:

```
CUST_NAME = 'ООО "Alpha"'
```

Следует отметить, что когда фильтр задается не в инспекторе объектов, а программно, то следует использовать функцию QuotedStr на тот случай, если в строке окажутся символы одинарной кавычки:

```
Table1.Filter:='CUST_NAME = ' + QuotedStr(str);
```

Если же сам фильтр вводится в код программы как литерал, то можно заранее продублировать кавычки:

```
Table1.Filter:='CUST_NAME = ''ООО "Gamma"''';
```

Теперь создадим небольшое приложение, которое будет осуществлять фильтрацию записей в наборе данных. В качестве набора вновь используем таблицу (Table), так же нам понадобятся компоненты DBGrid и DataSource. Для управления фильтром поместим на форму компонент-редактор (Edit) и 2 кнопки (Button). Первая кнопка, назовем ее «Применить» будет включать фильтрацию путем присвоения значения, определенного в редакторе, свойству Filter таблицы и назначения значения истины свойству Filtered:

```
Table1.Filter:=Edit1.Text;
Table1.Filtered:=true;
```

Вторая же кнопка – «Сброс» – будет просто отключать фильтрацию:

```
Table1.Filtered:=false;
```

Теперь остается запустить приложение, ввести какое-либо подходящее выражение в строку фильтра и нажать кнопку «Применить».

Исходный код примера находится в каталоге *Demo\Part4\Filter*.

Еще одним свойством набора данных, имеющим отношение к фильтрации, является FilterOptions. Оно представляет собой набор из 2 флагов – foCaseInsensitive и foNoPartialCompare. Первый из них, будучи установленным, делает фильтрацию в строках регистронезависимой. Второй же заставит интерпретировать знак * (звездочку) как символ, в противном случае звездочка будет выполнять роль шаблона подстановки. Например, можно определить фильтр таким образом:

```
CUST_NAME = 'ООО *'
```

В том случае, если флаг foNoPartialCompare выключен, то будут отобраны все записи, начинающиеся с «ООО ». Если же этот флаг включен, то подойдет только запись, имеющая именно такое значение, т.е. «ООО *».

Поиск

Помимо навигации и фильтрации, для работы с источниками данных применяется поиск. Подобно фильтрации, поиск заключается в нахождении полей, соответствующих заданным критериям. А при совпадении условий поиска с данными, может происходить перемещение курсора на первое подходящее поле – подобно навигации.

Для поиска по любым полям используют функции Locate и Lookup. При этом для поиска с переходом курсора на совпавшую запись используют функцию Locate, а для простого считывания данных из совпавшей записи – Lookup.

Функция Locate производит поиск записи с удовлетворяющими условия поиска значениями полей. Если такая запись будет найдена, то курсор будет переведен на нее, а функция возвратит истину. Если же совпадений не найдется, то функция возвратит ложь. Определена эта функция следующим образом:

```
function Locate(const KeyFields: string; const KeyValues: Variant; Options:
TLocateOptions): Boolean;
```

Здесь KeyFields определяет поля, по которым производится поиск, KeyValues - выражение для поиска, а Options задает такие параметры, как распознавание регистра в строках и возможность совпадения по части слова.

Для примера создадим приложение, которое сможет осуществлять поиск по названиям клиентов в нашей таблице Customers. Как обычно, понадобятся связка из компонентов Table, DataSource и DBGrid. Кроме них, задействуем однострочный редактор, кнопку и 2 переключателя типа CheckBox. Первый переключатель подпишем как «Без учета регистра», а второй – «Совпадение по части». Теперь остается написать код для процедуры обработки нажатия на кнопку, подобно тому, что приведен в листинге 20.3.

Листинг 20.3. Поиск с опциями

```
procedure TForm1.Button1Click(Sender: TObject);
var
 lo: TLocateOptions;
begin
 lo:=[];
 if CheckBox1.Checked then lo:=lo+[loCaseInsensitive];
 if CheckBox2.Checked then lo:=lo+[loPartialKey];
 Table1.Locate('CUST_NAME',Edit1.Text,lo);
end;
```

Данный пример иллюстрирует типичное применение поиска по одному полю (см. также пример в *Demo\Part4\Search*). В том же случае, если поиск необходимо провести по нескольким полям, то сами поля перечисляют через точку с запятой для параметра KeyFields и определяют массив значений для KeyValues:

```
Table1.Locate('CUST_NAME;CUST_ID',VarArrayOf(['ООО "Beta"',2]),lo);
```

Здесь производится поиск по полям CUST_NAME и CUST_ID, при этом совпавшими будут считаться записи, у которых в поле CUST_NAME будет значение «ООО "Beta"», а в поле CUST_ID – число 2. Что касается функции VarArrayOf, то она используется для одновременного создания и заполнения массива типа Variant.

Второй метод нахождения записей – Lookup – предназначен для нахождения искомого в источнике данных и возвращения нужных данных. Объявлена она следующим образом:

```
function Lookup(const KeyFields: string; const KeyValues: Variant; const
ResultFields: string): Variant;
```

Первые 2 аргумента аналогичны тем, что имеются в функции Locate, а 3-й используется для того, чтобы определить поля, значения которых необходимо получить. Сами значения возвращаются либо в виде значения какого-либо простого типа, либо в виде массива, если для возврата запрашивается более одного поля.

Помимо того, что функция Lookup не перемещает курсор на найденное поле, она всегда работает в режиме «строгого» поиска, т.е. при отсутствии опций, понимается, что регистр надо различать, а частичное совпадение недопустимо.

Функции Locate и Lookup хороши тем, что могут использоваться для поиска по любым полям набора данных. Если же поиск требуется осуществлять по индексированным полям, то выбор методов существенно расширяется. В частности, для поиска одной записи можно использовать методы GotoKey и FindKey, производящие поиск по точному соответствию. А методы GotoNearest и FindNearest производят поиск по частичному соответствию. При этом перед вызовом методов GotoKey и GotoNearest необходимо вызывать метод EditKey или SetKey, чтобы перевести компонент Table в режим редактирования ключа поиска:

```
Table1.EditKey;
Table1.FieldByName('CUST_NAME').AsString := Edit1.Text;
if not Table1.GotoKey then ShowMessage('Совпадения не найдено!');
```

Здесь подразумевается, что поле CUST_NAME является индексированным.

Еще одна группа методов, работающих с индексированными полями, позволяет работать с диапазоном записей. К ним относятся: SetRangeStart, SetRangeEnd, EditRangeStart, EditRangeEnd, ApplyRange и CancelRange. Для использования поиска диапазона записей необходимо установить начало и конец диапазона вызовом функций SetRangeStart и SetRangeEnd, или EditRangeStart и EditRangeEnd, указывая при этом граничные значения полей. Затем, вызвав метод ApplyRange, указанный диапазон применяется к набору данных:

```
with Table1 do begin
  SetRangeStart; //включаем режим установки начала диапазона
  FieldByName('CUST_ID').AsString:=Edit1.Text; //определяем начало диапазона
  SetRangeEnd; //включаем режим установки конца диапазона
  FieldByName('CUST_ID').AsString:=Edit2.Text; //определяем конец диапазона
  ApplyRange; //Применяем диапазон
end;
```

Результатом применения диапазона является набор данных, состоящий из удовлетворяющих условиям рядов. С этой точки зрения поиск диапазона очень похож на фильтрацию. Так же не следует забывать, что поля, участвующие в определении границ диапазона, должны быть индексированными.

Редактирование

Помимо поиска, фильтрации и навигации, существует другая не менее важная группа действий, выполняемых с наборами данных. Это редактирование (изменение), удаление и добавление записей. Все эти операции доступны только в том случае, если источник данных, во-первых, поддерживает модификацию данных как таковую, а во-вторых, если она разрешена в данный момент времени. В частности, компонент Table поддерживает модификацию, а за возможность ее производить отвечает свойство ReadOnly: если оно установлено в истину, то править данные нельзя, а если в ложь (что принято по умолчанию) – то можно. Проверить, допустимо ли изменение данных, можно при помощи свойства CanModify, доступное во время выполнения. И если оно содержит значение истины, то можно смело приступать к правке.

ПРИМЕЧАНИЕ

Следует отметить, что предоставить пользователю возможность правки данных возможно при помощи визуальных компонент – без написания какого-либо кода вообще, для чего достаточно воспользоваться компонентами DBGrid и DBNavigator.

Редактирование записей заключается в изменении значений их полей. Отредактирована в один момент времени может только одна запись, причем только текущая (та, на которую ссылается курсор). Поэтому перед редактированием следует выбрать запись, например, при помощи метода Locate. После этого последовательно выполняются следующие операции:

- перевод набора данных в режим редактирования (вызов метода Edit);
- собственно изменение значений полей записи;
- подтверждение изменений (метод Post) или отказ от них (метод Cancel).

В виде программного кода это реализуется следующим образом (см. пример в каталоге *Demo\Part4\Edit*):

```
Table1.Edit;
Table1.FieldByName('CUST_NAME').AsString:=Edit1.Text;
Table1.Post;
```

Здесь мы меняем значение поля CUST_NAME в текущей записи на содержимое однострочного редактора. Если бы требовалось изменить значение, скажем, числового поля, то следовало бы либо преобразовать значение Edit1.Text в число, либо использовать другой компонент, например, UpDown:

```
Table1.FieldByName('ANY_INT_VALUE').AsInteger:=UpDown1.Position;
```

Вместе с тем, если используются DB-ориентированные версии компонент, связанные с текущим набором данных, то метод Edit вызывать не придется, т.к. он будет вызван косвенно самим компонентом. Собственно, в таком случае, как это уже рассматривалось в предыдущей главе, вообще никаких действий предпринимать не придется, более того, если набор данных не поддерживает редактирование, то связанные с ним компоненты не дадут даже возможности попытаться изменить данные, уберегая и пользователя, и разработчика от возможных ошибок. С этой точки зрения для последовательного редактирования единичных записей предпочтительнее использовать именно такую, полностью автоматическую связку. Что касается программного изменения, то оно может оказаться очень полезным, а подчас и незаменимым, например, при обработке массивов данных.

Предположим, что у нас в таблице счетов хранятся суммы в евро. Но в какой-то момент потребовалось перевести их в доллары по текущему курсу. Соответственно, чтобы выполнить обработку всех записей, можно написать следующий код:

```
with Table1 do begin
  First;
  while not Table1.Eof do begin
    Edit;
    FieldByName('BILL_SUMM').AsCurrency:=FieldByName('BILL_SUMM').AsCurrency
*1.2;
    Post;
```

```
    Next;
  end;
end;
```

Обратите внимание, что метод Edit вызывается при каждой итерации цикла, поскольку после обработки метода Post таблица возвращается к режиму чтения. Так же полезно знать, что метод Next, в случае необходимости, может сам вызывать метод Post, однако если вместо Next использовать, скажем, FindNext, то Post вызван не будет. Поэтому, во избежание возможных ошибок в процессе модернизации кода, желательно вызывать метод Post явно.

Добавление и удаление

Подобно редактированию, добавление и удаление записей возможно только для модифицируемых наборов данных. В частности, для добавления новой записи следует выполнить следующую последовательность действий:

- перевести набор данных в режим вставки записи;
- определить значения полей новой записи;
- подтвердить сделанные изменения или отказаться от них.

Очевидно, что последовательность действий практически полностью аналогично тому, что мы уже видели при редактировании, с той лишь разницей, что вместо перехода к режиму редактирования используется переход в режим вставки. Для этих целей предусмотрено 4 метода – Insert, Append, InsertRecord и AppendRecord.

Методы Insert и Append переводят набор данных в режим вставки, и добавляют к нему новую пустую запись. Различие между этими методами состоит лишь в том, что если метод Append всегда добавляет запись в конец набора данных, то метод Insert – в позицию, на которой находится курсор ввода. Поэтому перед обращением к методу Insert обычно предварительно перемещают указатель в требуемую позицию набора данных. Например, для вставки нового ряда в самое начало набора, можно использовать следующий код:

```
Table1.First;
Table1.Insert;
```

После того, как набор данных будет переведен в режим добавления записи, остается произвести действия по определению значений полей, подобно тому, как это делается при редактировании. Фактически, это и есть редактирование, поскольку мы уже имеем готовый ряд полей, которые надо просто заполнить нужными значениями:

```
Table1.FieldByName('BILL_CUST').AsInteger:='5';
Table1.FieldByName('BILL_SUMM').AsCurrency:='155.99';
Table1.Post;
```

Если в программе были определены переменные, представляющие собой поля, то, разумеется, будет проще именно их и задействовать:

```
Table1BILL_CUST.AsInteger:='5';
Table1BILL_SUMM.AsCurrency:='155.99';
```

Более того, некоторые поля могут являться заполняемыми автоматически, например, поле BILL_ID в таблице счетов имеет автоинкрементный тип и получит нужное значение путем встроенных механизмов СУБД. Правда, произойдет это не сразу, а лишь

при выполнении последней стадии редактирования – подтверждения, что, как и при обычной правке делается методом Post. Если же отказаться от внесенной правки (метод Cancel), то не будут сохранены не только введенные значения, но и сам добавленный ряд тоже.

В ряде случаев, особенно когда требуется изменять большинство полей записи, удобно пользоваться специальным методом – SetFields. С его помощью можно задать значения для произвольного количества полей записи одновременно. Так, предыдущий пример с изменением 2 полей можно написать так:

```
Table1.SetFields([Nil, 5, 155.99]);
```

Здесь в качестве значения для первого поля указано значение Nil, поскольку подразумевается, что это поле автоинкрементного типа и значение для него будет назначено автоматически во время подтверждения. Отметим так же, что этого самого подтверждения (использования метода Post) в данном случае явно указывать не надо, поскольку метод SetFields вызывает его самостоятельно. Метод SetFields можно использовать не только при добавлении, но и при редактировании записей. В таком случае для тех полей, значения которых менять не требуются, так же указывают Nil.

Наконец, наиболее комплексным подходом к вставке записей является использование методов InsertRecord и AppendRecord. Метод InsertRecord объединяет в себе методы Insert и SetFields. Соответственно, чтобы добавить новую запись с его помощью, достаточно написать:

```
Table1.InsertRecord([Nil, 5, 155.99]);
```

Аналогично, метод AppendRecord, объединяет в себе методы Append и SetFields, т.е. добавляет и заполняет значениями запись в конце набора данных.

Что касается удаления записей, то для этих целей используется метод Delete. Он удаляет текущую запись из набора данных и перемещает указатель текущей записи на следующую запись. При этом если запись в момент вызова этого метода находилась в режиме вставки, то вызов Delete, по сути, будет аналогичен обращению к методу Cancel. Если же набор данных не содержит ни одной записи, то вызов метода Delete инициирует исключение.

Тот факт, что при удалении записи указатель перемещается автоматически, делает ненужным перемещение указателя методами типа Next. Например, чтобы удалить все записи из набора достаточно написать такой цикл:

```
while not Table1.Eof do Table1.Delete;
```

Поскольку на практике довольно редко приходится удалять все данные из таблицы БД, то удаление данных чаще всего совмещают с фильтрацией, т.е. сначала отбирают ставшие ненужными данные при помощи фильтра, а затем удаляют их, после чего отключают фильтрацию.

Простое приложение БД

На основе полученных знаний в области баз данных, создадим приложение, которое будет работать с нашей базой данных. Оно должно будет производить такие действия, как вывод списка клиентов с возможностью поиска по нему и выводить сведения обо всех счетах на отдельно взятого клиента. Так же предусмотрим возможность создания новых клиентов и счетов путем заполнения специальных форм. Назовем мы создаваемую программу «База 1.0».

Чтобы не отвлекаться на особенности визуального программирования, при разработке этого приложения мы постараемся максимально использовать программный код для выполнения всех действий, относящихся к взаимодействию с СУБД. Окончательный вариант того, что должно будет получиться, можно найти в каталоге *Demo\Part4\DBApp*.

Прежде чем приступить собственно к написанию приложения, немного модифицируем тестовую базу данных, для чего в таблицу счетов добавим поле даты счета (тип Date), а в таблицу клиентов – строковое поле адреса (модифицированный вариант БД находится в подкаталоге *data*, вложенном в указанный выше каталог с рассматриваемым приложением).

Теперь приступим к разработке самой «Базы 1.0». Для начала назовем главное окно приложения MainFrm и сохраним его под именем «main.pas», а сам проект сохраним под названием «mydb». Затем поместим на главной форме компонент DBGrid, отведя для него основное пространство окна, а под ним в ряд расположим однострочный текстовый редактор и 4 кнопки – «Баланс», «Новый счет», «Новый клиент» и «Выход» (рис. 20.2).

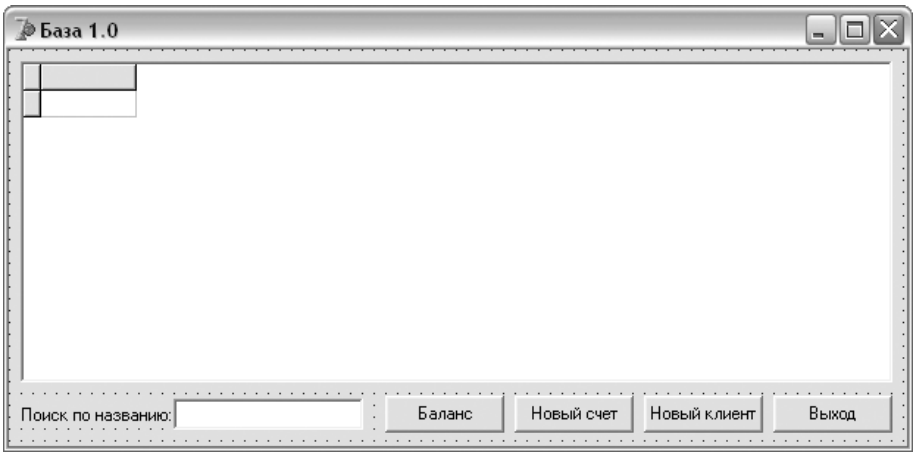

Рис. 20.1. Форма главного окна приложения

Чтобы в дальнейшем, при написании кода, не возникало вопросов, какой элемент интерфейса за что отвечает, присвоим всем компонентам более осмысленные имена. Так, таблицу назовем CustGrd, редактор – NameEd, а кнопки – BalanceBtn, NewBillBtn, NewCustBtn и CloseBtn. Учитывая тот факт, что для ввода новых клиентов мы собираемся предусмотреть отдельное окно, то чтобы придать приложению более «профессиональный» вид, в свойстве Options таблицы включим флаг dgRowSelect, в результате чего записи будут выделяться целиком. В свойствах самой формы отключим кнопку разворачивания на полный экран путем выключения флага biMaximize у свойства BorderIcons, а рамку сделаем неизменяемой, установив свойство BorderStyle в bsSingle. Наконец, для свойства Position выберем значение poScreenCenter, а в Caption напишем название программы – «База 1.0».

Дабы не загромождать главную форму приложения компонентами доступа к данным, используем форму данных (File → New → Data Module), на которую, в свою очередь, поместим компонент Database и 2 пары компонентов Table и DataSource. Компонент

Database назовем MainDB, это же значение используем как название базы данных (свойство DatabaseName), а для свойства DriverName выберем Standard. Компоненты-таблицы назовем CustTbl и BillTbl, а источники данных – CustBs и BillDs, после чего привяжем их к таблицам при помощи свойства DataSet. Саму форму данных назовем data, и сохраним файл под именем dm.pas.

Прежде, чем браться за дальнейшую работу, определим программный код, который будет выполнять собственно подключение к данным. Для этого нам потребуется определить параметры компонента Database (MainDB) и обеих таблиц в момент создания модуля. Сделать это можно, написав для события OnCreate формы data следующий код:

```
MainDB.Params.Add('path='+ExtractFilePath(paramstr(0))+'data');
MainDB.Connected:=true;
CustTbl.DatabaseName:='MainDB';
CustTbl.TableName:='customer.DB';
CustTbl.Active:=true;
BillTbl.DatabaseName:='MainDB';
BillTbl.TableName:='bill.DB';
BillTbl.Active:=true;
```

Теперь можно вернуться к главному окну приложения и после ключевого слова uses в части interface модуля добавить модуль dm. Это позволит нам уже на данном этапе указать в свойстве DataSource таблицы CustGrd нужное нам значение – data.CustDS.

Следующим этапом будет разработка формы отчета по счетам. Для этого нам понадобится еще одна форма (File → New → Form), на которой мы разместим таблицу DBGrid и одну единственную кнопку. Как и в главной форме, в части interface, в списке модулей после ключевого слова uses добавим dm. Теперь для свойства DataSource таблицы укажем значение data.BillDS, а саму таблицу назовем BillsGrd. Кроме того, подобно таблице в главном окне, включим флаг dgRowSelect в свойстве Options.

Единственную имеющуюся на этой форме кнопку назовем CloseBtn и сразу же напишем код для обработчика события OnClick, который будет состоять из единственного выражения: close. Наконец, свойство BorderStyle установим в bsDialog, а для свойства Position выберем значение poMainFormCenter.

Вновь перейдем к главному окну и напишем обработчик события для кнопки «Баланс», т.к. именно при помощи этой кнопки будет открываться окно счетов. Для того чтобы вывести все счета для выбранного клиента, нам понадобится фильтрация в таблице счетов по полю BILL_CUST. А для того, чтобы подсчитать сумму всех счетов, надо будет пройтись по всем отфильтрованным записям и просуммировать значение поля BILL_SUMM. Наконец, не помешает вывести в заголовок окна имя клиента. В результате мы получим код, приведенный в листинге 20.4.

Листинг 20.4. Вывод информации по счетам с подсчетом общей суммы

```
procedure TMainFrm.BalanceBtnClick(Sender: TObject);
var
 summ: Currency;
begin
 Data.BillTbl.Filter:=
'BILL_CUST='+Data.CustTbl.FieldByName('CUST_ID').AsString;
```

```
 Data.BillTbl.Filtered:=true;
 summ:=0;
 while not Data.BillTbl.Eof do begin
   summ:=summ+Data.BillTbl.FieldByName('BILL_SUMM').AsCurrency;
   Data.BillTbl.Next;
 end;
 BillsFrm.Caption:=Data.CustTbl.FieldByName('CUST_NAME').AsString+
': '+CurrToStr(summ);
 BillsFrm.ShowModal;
end;
```

Здесь сначала производится фильтрация таблицы счетов по текущему значению поля CUST_ID в таблице клиентов, после чего производится суммирование значений поля BILL_SUMM при помощи заранее объявленной переменной summ. После этого составляется заголовок окна, и оно показывается в качестве модального диалога. Если на данном этапе запустить приложение, то можно будет убедиться, что в главную форму выводится список клиентов, а при нажатии на кнопку «Баланс» открывается окно со счетами. Для упрощения навигации по списку клиентов (предположим, что их там у нас не несколько штук, а несколько десятков или сотен), реализуем поиск по вводу. Разумеется, подобный поиск должен быть регистро-независимым и осуществляться по части слова. Для этого в обработчике события OnChange текстового редактора NameEd следует предусмотреть соответствующий код:

```
Data.CustTbl.Locate('CUST_NAME',NameEd.Text,[loCaseInsensitive,loPartialKey]
);
```

Теперь займемся интерфейсом для создания новых счетов и новых клиентов. Для этого нам понадобятся еще 2 формы, назовем их BillFrm и CustFrm.На первой нам понадобятся компонент-редактор для ввода суммы (назовем его SummEd) и компонент DateTimePicker для указания даты (DatePick). На второй – 2 редактора: один – для ввода имени клиента (NameEd), другой – для адреса (AddrEd). Так же на обоих формах нам потребуются кнопки «ОК» и «Отмена». Поскольку оба этих окна, подобно окну со счетами, будут модальными диалогами, то установим свойства BorderStyle в bsDialog, и Position – в poMainFormCenter. Для кнопок отмены в обоих случаях код будет выглядеть следующим образом:

```
ModalResult:=mrCancel;
```

Это позволит определить дальнейшие действия программы после того, как окно закроется и выполнение должно будет быть продолжено в том месте, откуда данное окно было вызвано. Что касается кнопок ОК, то для формы нового счета она должна будет проверять на корректность значение, введенное в поле суммы. Реализовать такую проверку можно так:

```
if StrToCurrDef(SummEd.Text,0)<>0 then ModalResult:=mrOk
   else MessageDlg('Сумма накладной не может быть нулевой, либо число
введено неверно',mtError,[mbOk],0);
```

Здесь подразумевается, что по логике работы программы сумма счета не может быть равной нулю. Таким образом, если пользователь попробует создать нулевой счет, либо в поле суммы напишет что-то, что невозможно преобразовать в вещественное число, то он получит соответствующее сообщение. Если же все указано верно, то окно закроется и вызвавшая его функция получит значение mrOk.

Аналогичным образом следует организовать проверку и в окне клиента, с той лишь разницей, что проверять в нем следует содержимое строки с названием – оно не должно быть пустым:

```
if NameEd.Text<>'' then ModalResult:=mrOk
   else MessageDlg('Не указано имя клиента',mtError,[mbOk],0);
```

Собственно, остается только написать код, который и будет вызывать эти окна из главной формы, и производить нужные манипуляции над данными. В частности, для создания нового счета нам надо будет показать окно-форму счета, не забыв при этом указать в заголовке окна имя клиента, затем, если форма возвратит результат mrOk, то выполнить вставку данных, введенных пользователем в окне счета, в таблицу bill. Такая процедура, написанная с использованием метода Append, представлена в листинге 20.5. В нем же представлен вариант добавления записи в таблицу клиентов, но с использованием комбинированного метода AppendRecord.

Листинг 20.5. Добавление новых записей в таблицу счетов и в таблицу клиентов

```
procedure TMainFrm.NewBillBtnClick(Sender: TObject);
begin
 with BillFrm, Data.BillTbl do begin
   Caption:='Счет для '+Data.CustTbl.FieldByName('CUST_NAME').AsString;
   ShowModal;
   if ModalResult<>mrOk then exit;
   Append;
   FieldByName('BILL_CUST').AsInteger:=
Data.CustTbl.FieldByName('CUST_ID').AsInteger;
   FieldByName('BILL_SUMM').AsString:=SummEd.Text;
   FieldByName('BILL_DATE').AsDateTime:=DatePick.Date;
   Post;
 end;
end;
procedure TMainFrm.NewCustBtnClick(Sender: TObject);
begin
 with CustFrm do begin
   ShowModal;
   if ModalResult<>mrOk then exit;
   Data.CustTbl.AppendRecord([nil,NameEd.Text,AddrEd.Text]);
   Data.CustTbl.Post;
 end;
end;
```

Таким образом, мы реализовали все задуманные функции программы. Единственным недостатком на данный момент является то, что программа не предоставляет интерфейс для изменения данных. Однако это довольно просто реализуется путем использования тех же форм, что мы уже создали для добавления данных. Все отличия будут состоять лишь в том, что при редактировании они должны будут появляться с уже заполненными полями, а при подтверждении изменений пользователем вместо добавления новой записи будет производиться модификация данных в текущей записи. Само же отображение формы логично будет ассоциировать с двойным щелчком

мышкой по соответствующей таблице. Так, для редактирования записи клиента, можно определить следующий обработчик события OnDblClick таблицы CustGrd:

```
with CustFrm do begin
  NameEd.Text:=Data.CustTbl.FieldByName('CUST_NAME').AsString;
  AddrEd.Text:=Data.CustTbl.FieldByName('CUST_ADDRESS').AsString;
  ShowModal;
  if ModalResult<>mrOk then exit;
  Data.CustTbl.Edit;
  Data.CustTbl.FieldByName('CUST_NAME').AsString:=NameEd.Text;
  Data.CustTbl.FieldByName('CUST_ADDRESS').AsString:=AddrEd.Text;
  Data.CustTbl.Post;
end;
```

Что касается изменения счетов, то в форме списка счетов (BillsFrm) определим вызов формы составления счета (BillFrm) так же по двойному щелчку, по таблице BillsGrd. При этом следует не забыть указать модуль формы счета в списке uses части implementation. Полный код этой части модуля приведен в листинге 20.6.

Листинг 20.6. Часть implementation модуля bills

```
implementation
uses bill;
{$R *.dfm}
procedure TBillsFrm.CloseBtnClick(Sender: TObject);
begin
 close;
end;
procedure TBillsFrm.BillsGrdDblClick(Sender: TObject);
begin
 with BillFrm, Data.BillTbl do begin
   Caption:='Счет для '+Data.CustTbl.FieldByName('CUST_NAME').AsString;
   SummEd.Text:=FieldByName('BILL_SUMM').AsString;
   DatePick.Date:=FieldByName('BILL_DATE').AsDateTime;
   ShowModal;
   if ModalResult<>mrOk then exit;
   Edit;
   FieldByName('BILL_SUMM').AsString:=SummEd.Text;
   FieldByName('BILL_DATE').AsDateTime:=DatePick.Date;
   Post;
 end;
end;
end.
```

Теперь наша программа умеет даже несколько больше, чем планировалось ранее. А именно, помимо обеспечения удобной навигации по списку клиентов (под удобством мы здесь понимаем возможность поиска по первым введенным буквам), просмотра счетов для каждого клиента, и созданию новых записей клиентов и счетов, мы так же получили возможность изменять все эти записи. Теоретически, подобных функциональных возможностей мы могли бы добиться, просто поместив на форму 2 связанных по принципу «один ко многим» таблицы, как это уже было продемонстрировано

в самом начале этой части книги. Однако для конечного пользователя тот вариант был бы гораздо менее понятным и удобным. Кроме того, здесь мы реализовали базовую проверку вводимых значений на допустимость, что на практике позволяет минимизировать возможные ошибки при вводе данных.

Запросы

В предыдущих главах, посвященных работе с СУБД, мы рассмотрели базовые аспекты работы с данными с использованием средств VCL. Вместе с тем, при работе с СУБД очень часто возникает необходимость в выполнении более сложных операций. Например, это может быть сложная выборка данных по таким условиям, которые проблематично или вовсе невозможно определить при помощи фильтра – например, из нескольких таблиц одновременно. Для таких целей используют специальный язык, который «понимают» СУБД – язык SQL, а для создания обращений к СУБД на этом языке в VCL имеется специальный компонент – Query.

Язык SQL

Все современные клиент-серверные СУБД имеют одну общую черту – работа с ними строится на языке SQL – Structured Query Language (структурированный язык запросов). Кроме того, чтобы не лишать разработчиков БД на Delphi возможности работать при помощи SQL с обычными файловыми СУБД типа dBase и Paradox, BDE предоставляет возможность прозрачного использования SQL и для них, используя свои внутренние механизмы.

Стандарт языка SQL определяется ANSI (Американским национальным институтом стандартов). Вместе с тем, каждый производитель СУБД считает своим долгом дополнить язык собственными расширениями которые, как они считают, будут весьма полезны. Иногда они несколько нарушают стандарт языка, хотя хорошие идеи имеют тенденцию развиваться и вскоре становиться стандартами де-факто сами по себе в силу полезности своих качеств. Здесь мы будем рассматривать SQL на основе наиболее полно поддерживаемых всеми СУБД вариантами ANSI-89 и ANSI-92.

Язык SQL предназначен для манипулирования данными в реляционных базах данных, определения структуры баз данных и для управления правами доступа к данным в многопользовательской среде. По этой причине язык SQL состоит из 3 составных частей:

- Языка манипулирования данными (Data Manipulation Language, DML);
- Языка определения данных (Data Definition Language, DDL);
- Языка управления данными (Data Control Language, DCL).

Все эти языки являются составной частью языка SQL. Фактически, каждый из них содержит набор команд SQL, предназначенных для своей области. В процессе работы с БД, пожалуй, чаще всего используется язык манипулирования данными, состоящий из 4 основных команд – SELECT, INSERT, UPADTE и DELETE, при помощи которых производится, соответственно, выборка, вставка, обновление и удаление данных.

Язык определения данных служит для создания и изменения структуры БД – таблиц, индексов и т.д. Он состоит из 3 групп команд – CREATE, ALTER и DROP (создание,

изменение и удаление, соответственно), каждая из которых может манипулировать с одним из 6 объектов – базой данных, таблицей, виртуальной таблицей, индексом, триггером или хранимой процедурой. Таким образом, например, для команды CREATE мы получаем следующие 6 вариантов:

- CREATE DATABASE – создать базу данных;
- CREATE TABLE – создать таблицу;
- CREATE VIEW – создать виртуальную таблицу;
- CREATE INDEX – создать индекс;
- CREATE TRIGGER – создать триггер;
- CREATE PROCEDURE – создать хранимую процедуру.

Впрочем, триггеры и, в особенности, хранимые процедуры относятся исключительно к компетенции промышленных СУБД, которые мы в этой книге рассматривать не будем. То же самое относится и к языку управления данными (иногда его еще называют языком управления доступом) – он состоит из 2 основных команд – GRANT (дать права) и REVOKE (забрать права).

ПРИМЕЧАНИЕ

> На самом деле, для современных клиент-серверных СУБД существует ряд дополнительных команд SQL, позволяющих, например, определять собственные функции для обработки данных (CREATE FUNCTION), создавать резервные копии таблиц, преобразовывать типы таблиц и т.д. Далеко не все они входят в состав стандарта языка SQL, поэтому информацию по ним следует брать из поставляемой вместе с конкретной СУБД документации.

Мы не будем рассматривать точный синтаксис всех команд SQL, поскольку это потребовало бы объема, соизмеримого с целой книгой. Вместо этого мы рассмотрим основные команды на примерах, что намного более важно для понимания SQL, чем точный синтаксис, который, при необходимости, всегда можно посмотреть в документации на используемую СУБД. А начнем с рассмотрения команд языка манипулирования данными.

Команда SELECT

Наиболее важной командой языка манипулирования данными является команда SELECT (выбрать). За кажущейся простотой ее синтаксиса скрывается огромное число возможностей. В простейшем случае с ее помощью можно просто отобразить все содержимое таблицы. В других случаях при помощи одной команды SELECT можно одновременно производить такие операции, как фильтрация, сортировка по одной лил нескольким таблицам сразу. В самой обобщенной форме синтаксис этой команды можно представить следующим образом:

```
SELECT выражения_для_выборки FROM таблицы [параметры выборки]
```

В качестве выражения для выборки обычно перечисляют поля таблиц. Если требуется вывести все поля, то используют символ «*»:

```
SELECT BILL_ID, BILL_SUMM FROM BILL
SELECT * FROM BILL
```

В первом случае будут выведены поля BILL_ID и BILL_SUMM из таблицы BILL, а во втором – все поля из этой же таблицы. В этом нетрудно убедиться, воспользовавшись утилитой SQL Explorer для написания запроса. Для этого запустите SQL Explorer, откройте нужную БД, например, DATA1 (или предварительно создайте новый алиас для более полной версии БД, созданной для примера DBApp), и в правой части окна щелкните по закладке Enter SQL, после чего введите нужный код и нажмите на кнопку Execute Query. Результат выполнения запроса незамедлительно будет выведен в таблицу внизу (рис. 21.1).

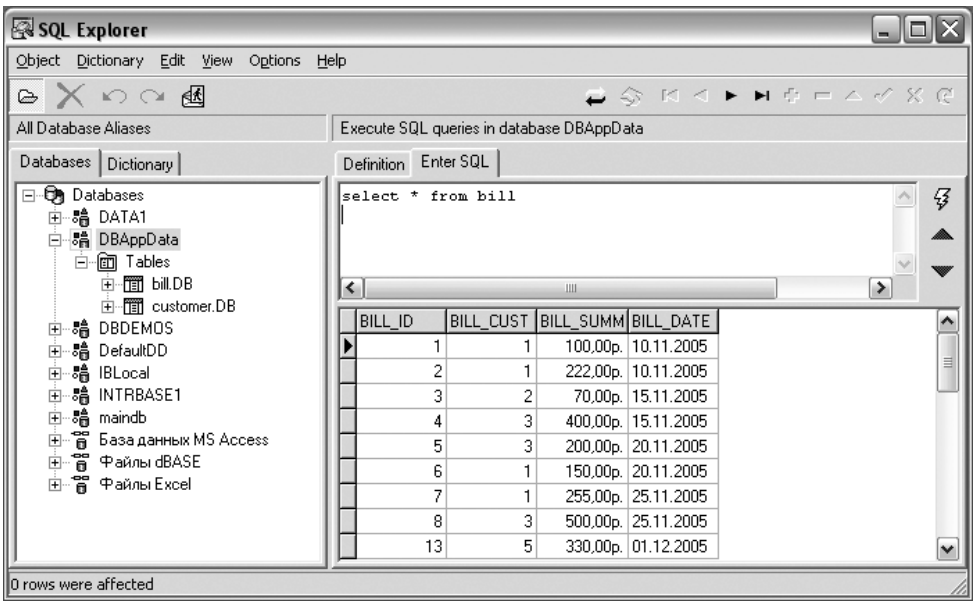

Рис. 21.1. Приложение SQL Explorer

СОВЕТ

> Если у вас установлена версия Delphi, не имеющая этого инструмента, то для экспериментов с командой SELECT вы можете использовать приложение SQLELE, которое можно найти в каталоге *Tools\SQLE_LE*.

При выборе можно сразу же произвести упорядочивание записей, причем SQL позволяет делать эту операцию по любому полю таблицы, а не только по индексированному. Например, чтобы вывести все счета в порядке возрастания суммы, достаточно написать следующее выражение:

```
SELECT * FROM bill ORDER BY BILL_SUMM
```

Если бы требовалось выполнить сортировку не по возрастанию, а по убыванию, то в конце выражения следовало бы добавить ключевое слово DESC:

```
SELECT * FROM bill ORDER BY BILL_SUMM DESC
```

Выборка записей из таблицы производится при помощи условия, определяемого после ключевого слова WHERE. Например, если надо выбрать только те счета, которые были после 15 декабря, следует написать следующий SQL-запрос (формат написания даты в данном случае зависит от системных установок):

```
SELECT * FROM bill WHERE BILL_DATE>'15.12.2005'
```

Если при этом так же требуется еще и выполнить сортировку, то в конец выражения добавляют определение ORDER BY:

```
SELECT * FROM bill WHERE BILL_DATE>'15.12.2005' ORDER BY BILL_SUMM
```

В случае, когда условий больше одного, то их объединяют при помощи логических AND или OR:

```
SELECT * FROM bill WHERE BILL_DATE>'15.12.2005' AND BILL_SUMM>100
SELECT * FROM bill WHERE BILL_DATE>'15.12.2005' OR BILL_SUMM>300
```

Все эти примеры иллюстрируют общую форму команды SELECT в языке SQL (для одной таблицы): SELECT (выбрать) указанные поля FROM (из) указанной таблицы WHERE(где) заданные условия истинны.

Немного более сложным вариантом является выборка данных из 2 и более таблиц сразу. Например, для наглядности не помешало бы вывести в виде таблице список счетов, содержащий не внутренние номера клиентов, а их имена. В этом случае придется задействовать 2 таблицы – bill и customer, причем связующим полем будет номер клиента, т.е. поле BILL_CUST в таблице счетов и поле CUST_ID в таблице клиентов. В результате, мы получаем следующее выражение:

```
SELECT CUST_NAME, BILL_SUMM FROM bill, customer WHERE CUST_ID = BILL_CUST
```

Опять-таки, к данному условию, связывающему поля таблиц, можно добавить дополнительные, скажем, ограничивающие даты или суммы, равно как и определить упорядочивание вывода. Например, можно добавить условие, что сумма счета должна быть свыше 250, а так же задать упорядочивание по имени клиента (рис. 21.2).

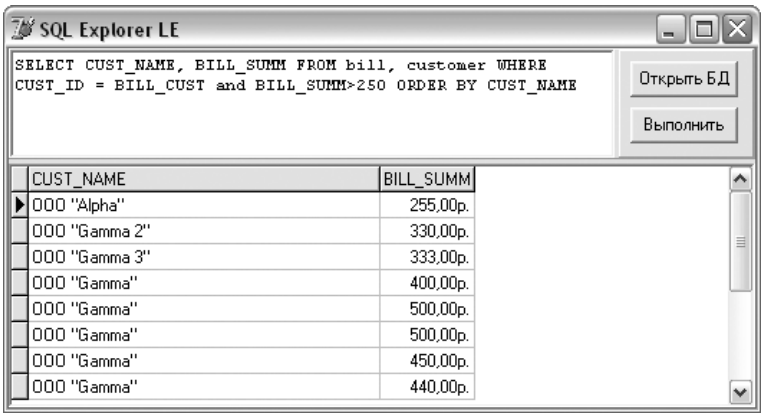

Рис. 21.2. Результат выполнения запроса по 2 таблицам

Следует учитывать, что подобное связывание таблиц – путем указания условия в конструкции WHERE, является устаревшим и применяется лишь тогда, когда требуется добиться совместимости с SQL89. Более новый стандарт, SQL92, рекомендует использовать для связывания таблиц ключевое слово JOIN. Таким образом, предыдущий пример можно переписать следующим образом:

```
SELECT CUST_NAME, BILL_SUMM FROM customer JOIN bill ON CUST_ID = BILL_CUST
```

Важно отметить, что условия объединения помещают после ключевого слова ON, в то время, как остальные условия можно указать после WHERE. Таким образом, если

бы нам надо бы добавить ограничения на сумму, то можно было бы написать 2 равнозначных варианта:

```
SELECT CUST_NAME, BILL_SUMM FROM customer JOIN bill ON CUST_ID = BILL_CUST
AND BILL_SUMM>250;
```

```
SELECT CUST_NAME, BILL_SUMM FROM customer JOIN bill ON CUST_ID = BILL_CUST
WHERE BILL_SUMM>250;
```

Для удобства можно задавать локальные «имена» как для полей, так и для таблиц. Это бывает полезным в тех случаях, когда в большом и сложном запросе используется множество условий с длинными именами полей и таблиц:

```
SELECT EXTRA_USER_DATA_INFO AS a, EXTRA_USER_DATA_NAME as b FROM USER_TABLE
WHERE a > 10 AND a < 100 AND b != 'Super' ORDER BY b
```

Следует отметить, что при выводе полей таблиц с помощью SQL можно производить некоторые вычисления. В этих случаях использование локальных имен бывает просто необходимым. Допустим, в счетах у нас хранятся суммы без НДС. В таком случае мы можем ввести сразу 2 столбца с суммами – с той, что хранится таблице и в виде вычисленного значения.

```
SELECT BILL_DATE, BILL_SUMM, BILL_SUMM * 1.18 as WITH_TAX FROM BILL ORDER BY
WITH_TAX
```

В таком случае названием столбца с вычисленным значением будет «WITH_TAX» (см. рис. 21.3).

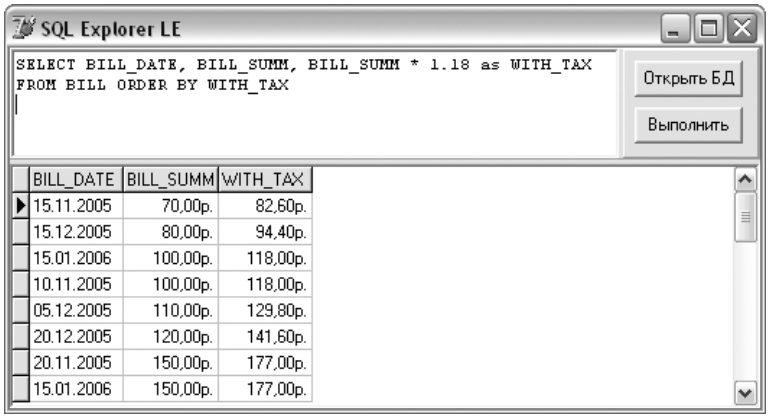

Рис. 21.3. Запрос с вычислениями

При подобном переименовании столбцов следует быть внимательным, поскольку использование идентификаторов, совпадающих с ключевыми словами SQL, приведет к ошибке. Кроме того, следует избегать использования символов кириллицы, поскольку они не поддерживаются многими СУБД в данном контексте.

Если написание арифметического выражения приведет к изменению выводимых данных в записях столбца, то использование агрегирующих (агрегатных) функций поможет выполнить действия над всеми записями, удовлетворяющими указанным в запросе условиям. Пожалуй, чаще всего используется функция COUNT, позволяющая узнать количество записей. Например, чтобы узнать, сколько записей хранится в таблице счетов, достаточно написать следующий запрос:

```
SELECT COUNT(*) FROM bill
```

Результатом выполнения этого запроса будет число записей в таблице счетов. Если бы мы хотели узнать число записей, сделанных, скажем, до 1января, то могли бы написать такое выражение:

```
SELECT COUNT(*) FROM bill WHERE BILL_DATE < '01.01.2006'
```

Помимо COUNT, в стандартном SQL предусмотрено еще 4 агрегирующих функции – для вычисления суммы (SUM), максимального (SUM) и минимального (MIN) значений столбцов, а так же среднего арифметического (AVG). Результатом выполнения запроса по всем этим функциям для таблицы счетов будет таблица, состоящая из 4 столбцов, в каждом из которых будет находиться результат вычислений (рис. 21.4).

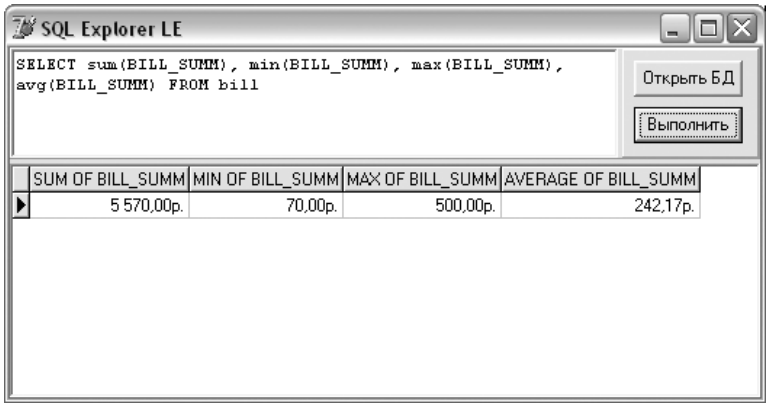

Рис. 21.4. Результат выполнения запроса с агрегирующими функциями

Что касается заданий ограничений на обрабатываемые данные, ровно, как и на выводимые при помощи «обычного» запроса строки, то помимо реляционных операторов сравнения – больше (>), меньше (<), равно (=), не равно (!= или <>), больше или равно (>=), меньше или равно (<=), – можно использовать следующие операторы:

- BETWEEN – задает диапазон значений, для которого выражение принимает истину;
- IN – проверяет, входит ли заданное значение, предшествующее ключевому слову «IN» (например, значение столбца) в указанный в скобках список;
- LIKE – проверяет, соответствует ли данное символьное значение строке с указанной маской (только для символьных данных);

В качестве примеров можно привести следующие варианты SQL-запросов:

```
SELECT BILL_SUMM FROM bill WHERE BILL_SUMM BETWEEN 100 and 200; /* Вывести
счета с суммами от 100 до 200 */

SELECT BILL_SUMM, BILL_CUST FROM bill WHERE BILL_SUMM IN (100,150,200) /*
Вывести счета с суммами, равными 100, 150 и 200 */

SELECT CUST_NAME FROM customer WHERE CUST_NAME LIKE 'ООО "Gamma%' /* Вывести
клиентов, имя которых начнается с «ООО "Gamma» */

SELECT CUST_NAME FROM customer WHERE CUST_NAME LIKE 'ООО "_____"' /* Вывести
клиентов, имя которых начнается с «ООО "» и имеет 5 символов в кавычках */

SELECT CUST_NAME FROM customer WHERE CUST_NAME LIKE '_%' /* Вывести
клиентов, имя которых состоит хотя бы из 1 символа */
```

Что касается порядка вывода результатов, то помимо сортировки, еще одним способом повлиять на вывод результатов запроса является использование GROUP BY. Эта конструкция определяет порядок группировки полей при выводе. При этом следует учитывать, что должны быть указаны все поля, перечисленные в запросе:

```
SELECT BILL_SUMM, BILL_DATE FROM bill GROUP BY BILL_DATE, BILL_SUMM
```

В данном случае данные будут сгруппированы по датам. Впрочем, это не самый удачный пример, поскольку такого же результата можно было бы добиться простой сортировкой по полю BILL_DATE. Поэтому рассмотрим более сложный запрос, который будет выводить общие суммы всех счетов для каждого из клиентов. Для этого нам понадобится агрегирующая функция SUM:

```
SELECT BILL_CUST, SUM(BILL_SUMM) FROM BILL GROUP BY BILL_CUST
```

Наконец, усложним запрос, сделав результаты его работы более наглядными. Для этого нам потребуется связать таблицу счетов с таблицей клиентов, а заодно произвести сортировку по итоговому результату. Таким образом, мы получим следующий запрос:

```
SELECT CUST_NAME, SUM(BILL_SUMM) as TOTAL FROM BILL JOIN CUSTOMER ON
CUST_ID=BILL_CUST GROUP BY BILL_CUST, CUST_NAME ORDER BY TOTAL DESC
```

Результатом его выполнения будет таблица, содержащая в первом столбце имена клиентов, а во втором – суммы всех их счетов, по которой и была произведена сортировка (рис. 21.5).

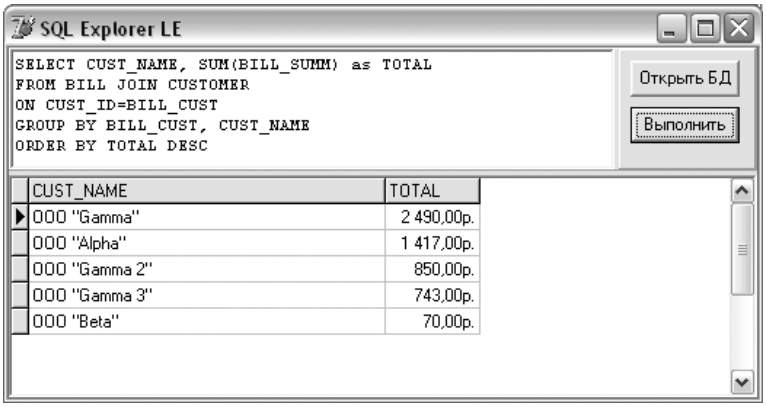

Рис. 21.5. Запрос с группировкой, агрегацией и упорядочиванием по вычисляемому полю

Все, что было на данный момент рассказано про команду SELECT, на самом деле, является лишь верхушкой айсберга ее возможностей. В то же время, даже такое беглое знакомство дает понять, что без знания SQL написать достаточно объемное приложение для БД крайне затруднительно. В то же время, каждая СУБД, как уже было отмечено, имеет свои тонкости в реализации языка, и команда SELECT, по причине своих огромных возможностей, является одним из наиболее частых претендентов на «модернизацию». По этой причине для дальнейшего изучения как команды SELECT, так и прочих возможностей SQL, следует использовать документацию к используемой СУБД.

Некоторые другие команды SQL

Помимо уже рассмотренных запросов на выборку данных при помощи команды SELECT, рассмотрим некоторые другие команды языка манипулирования данными. В частности, это команды INSERT (вставить), UPDATE (обновить) и DELETE (удалить).

Начнем со вставки – команды INSERT, которая добавляет запись с указанными значениями полей в конец таблицы. В упрощенной форме основной синтаксис этой команды можно представить следующим образом:

```
INSERT INTO таблица [(поля)] VALUES (значения)
```

В простейшем случае, когда надо заполнить все (или почти все) поля новой записи, достаточно написать подобное выражение:

```
INSERT INTO regions VALUES (77,'Москва')
```

В том же случае, если требуется указать значения только для определенных полей, то их предварительно перечисляют после имени таблицы:

```
INSERT INTO bill (BILL_CUST, BILL_SUMM, BILL_DATE) VALUES (1, 100,
'01.01.2006')
```

В принципе, то, что выполняется при помощи команды SQL INSERT, похоже на то, что делает метод InsertRecord. Однако SQL предоставляет дополнительные возможности, в частности, вставку целой группы записей:

```
INSERT INTO bill (BILL_CUST, BILL_SUMM, BILL_DATE) VALUES (1, 100,
'01.01.2006'), (1,200,'01.02.2006'), (1,150,'01.03.2006')
```

Еще более широкие возможности открываются при помощи комбинирования команды INSERT с командой SELECT, причем вставку можно производить как в ту же самую таблицу, так и в любую другую. Например, если нам надо в некую таблицу bill2 внести все записи из таблицы bill, имеющие суммы свыше 300, мы можем написать следующий запрос

```
INSERT INTO bill2 SELECT * FROM bill WHERE BILL_SUMM>300
```

Здесь предполагается, что структура таблиц bill и bill2 полностью идентична. Если же это было бы не так, то можно внести только часть полей (при условии, что типы полей будут совпадать). Например, если в таблице bill2 было бы поле для суммы, то для вставки всех подходящих сумм можно написать следующий запрос:

```
INSERT INTO bill2 (B2_SUMM) SELECT BILL_SUMM FROM bill WHERE BILL_SUMM>300
```

ПРИМЕЧАНИЕ

> При помощи INSERT-SELECT можно очень просто копировать данные из одной таблицы в другую. В то же время, если надо просто вставить данные из подготовленного текстового файла, то ряд СУБД предлагает более удобные и быстрые команды, например, LOAD DATA INFILE в MySQL.

Другая команда SQL, применяемая для модификации данных – это UPDATE. Она позволяет изменять значения в столбцах данных таблицы. В общих чертах ее синтаксис можно представить следующим образом:

```
UPDATE таблица SET имя_поля = значение [WHERE условие]
```

В простейшем случае, когда надо изменить все записи в колонке таблицы, можно написать следующее выражение:

```
UPDATE bill SET BILL_DATE = '01.01.2006'
```

Можно так же указывать не абсолютные, а вычисляемые значения:

```
UPDATE bill SET BILL_SUMM = BILL_SUMM * 2
```

Но чаще всего все-таки указывают условия, по которым следует находить те записи, которые следует изменить. Например, чтобы пересчитать все счета, выставленные до 31 января 2005 года, можно написать следующее выражение:

```
UPDATE bill SET BILL_SUMM = BILL_SUMM + 10 WHERE BILL_DATE<'31.12.2005'
```

Наконец, когда требуется изменить лишь какую-то определенную запись, что в качестве условия достаточно указать уникальное поле. Например, для таблицы счетов это поле BILL_ID, по которому можно однозначно идентифицировать какую-либо запись в таблице:

```
UPDATE bill SET BILL_SUMM = 1000 WHERE BILL_ID = 10
```

ВНИМАНИЕ

> Следует учитывать, что если предусмотренные в VCL методы для работы с БД, как правило, влияют лишь на одну запись (т.е. на ту, что выделена курсором), то в SQL изменению подвержены все записи указанной таблицы.

Наконец, остается рассмотреть последнее часто применяемое действие – удаление записей из таблицы. В SQL для этих целей используют команду DELETE, определенную следующим образом:

```
DELETE FROM таблица [WHERE условие]
```

Подобно команде UPDATE, если не указано условие, то удалению будут подвержены все данные в таблице. Например, чтобы удалить все записи в таблице счетов, достаточно написать:

```
DELETE FROM bill
```

В более вероятном случае требуется удалить лишь единичные записи. Например, удалению могут быть подвержены счета, выписанные до 2006 года, или же какой-либо определенный (ошибочный) счет:

```
DELETE FROM bill WHERE BILL_DATE<'01.01.2006'
DELETE FROM bill WHERE BILL_ID = 15
```

Таким образом, на текущий момент мы рассмотрели все основные команды SQL, относящиеся к языку манипулирования данными – SELECT, INSERT, UPDATE и DELETE. Во многих СУБД помимо этих команд имеется ряд дополнительных, например, TRUNCATE (аналог DELETE для всей таблицы) или REPLACE (вариация на тему вставки). Поэтому остается еще раз посоветовать ознакомиться с документацией на используемую СУБД.

То же самое касается и языка определения данных, представленного вариациями команд CREATE, DROP и ALTER: у каждой СУБД имеется собственный набор типов данных и свои тонкости работы с таблицами, индексами и т.д. В принципе, работая с «родными» для Delphi типами СУБД, вполне можно довольствоваться теми возможностями, что предоставляет утилита Database Desktop. Многие другие СУБД так же

располагают собственными средствами, упрощающими процесс создания таблиц, в качестве примера можно привести широко распространенную утилиту phpMyAdmin, обеспечивающую управление СУБД MySQL через веб-интерфейс.

Компонент запроса Query

До текущего момента мы рассматривали язык SQL как таковой, оставляя вопросы его взаимодействия с Delphi. Переходя к практическим вопросам использования этого языка в приложениях, рассмотрим компонент Query. Компонент Query представляет собой компонент набора данных, записи которого формируются в результате выполнения SQL-запроса. При этом текст запроса также содержится в этом компоненте в виде свойства SQL типа TStrings.

В целом, с точки зрения использования в приложении, компонент Query похож на другой компонент BDE – Table. Подобно Table, он так же может выступать в качестве источника данных. Однако благодаря тому, что практически все параметры, относящиеся к выборке данных, определяются в тексте запроса, то среди свойств этого компонента, связывающих его с БД, имеется лишь DatabaseName.

Еще одним важным отличием компонента Query от Table является отсутствие у Query свойства ReadOnly. Дело в том, что компонент Query по своей сути обычно предоставляет данные, доступные только для чтения, т.е. связь получается односторонняя. Этот факт следует учитывать, т.к. в ряде случаев при установке взаимодействий между компонентами бывает необходимым ссылаться на источники данных, поддерживающих непосредственную правку. В то же время, если выполняется ряд определенных условий, в частности, запрос обращается только к одной таблице, сама таблица поддерживает запись, а свойство RecuestLive компонента Query установлено в истину, то к такому запросу можно будет обращаться точно так же, как к таблице.

В то же время, подобные ограничение вовсе не говорит о том, что при помощи Query сложно изменять данные – просто для этих целей понадобится создавать соответствующий SQL-запрос (например, с командой UPDATE), а не пытаться использовать свойства объектов Filed. Кроме того, к Query неприменимы такие методы, как FindFirst, FindLast и т.д.

В качестве примера простейшего приложения, использующего этот компонент, рассмотрим уже упоминавшийся SQL Explorer LE. В каталоге *Tools\SQLE_LE*, помимо самого приложения, находится его исходный код. Не вдаваясь в подробности реализации, отметим лишь, что в приложении задействовано 3 компонента, связанных с доступом к данным – это Database, Query и DataSource. Еще один невизуальный компонент – диалог открытия файла используется для выбора каталога с БД. Весь написанный код этого приложения приведен в листинге 21.1.

Листинг 21.1. Исходный код SQLE LE

```
procedure TMainFrm.OpenBtnClick(Sender: TObject);
begin
 if not OpenDlg.Execute then exit;
 MainDB.Connected:=false;
 MainDB.Params.Clear;
 MainDB.Params.Add('path='+ExtractFilePath(OpenDlg.FileName));
 MainDB.Connected:=true;
```

```
 RunBtn.Enabled:=true;
end;
procedure TMainFrm.RunBtnClick(Sender: TObject);
begin
 Query1.Close;
 Query1.SQL.Text:=Memo1.Text;
 if pos('select',lowercase(Memo1.Text))=0 then Query1.ExecSQL else
 Query1.Open;
end;
```

Здесь кнопка открытия БД устанавливает путь к базе данных (подразумевается, что используется СУБД Paradox), после чего делает соединение активным, а кнопку выполнения запроса – доступной. Код для кнопки выполнения запроса нуждается в некотором пояснении: дело в том, что у компонента Query имеется 2 способа выполнения запроса – при помощи методов Open и ExecSQL. Принципиальная разница между ними состоит в том, что метод Open используется для запросов, производящих выборку данных (т.е. SELECT), в остальных же случаях предпочтительнее использовать метод ExecSQL.

Особенности работы с запросами

При рассмотрении компонента Query мы уже отметили ряд различий в подходах к использованию этого компонента по сравнению с Table. Однако этим различия в приложениях, основанных на SQL, не ограничиваются. Основным преимуществом запросов на SQL является то, что они позволяют минимизировать объем данных, которыми обмениваются СУБД и приложение. Например, когда из таблицы требуется отобрать какую-то часть записей, то в случае использования фильтрации или иных «обычных» методов, приложение запрашивает у СУБД всю таблицу. В том же случае, когда используется запрос, приложение получает лишь то, что ему требуется. Это существенным образом сказывается на быстродействии, особенно если обрабатываются большие объемы данных и при передаче информации по сети.

Для иллюстрации возможностей SQL и компонента Query модифицируем приложение «База 1.0» таким образом, чтобы оно использовало данные технологии. В частности, SQL может нам понадобиться для таких вещей, как вывод счетов для клиента, подсчет суммы счетов, внесение новых счетов и клиентов или редактирование имеющихся.

При разработке нового варианта приложения заменять без оглядки все компоненты Table на Query не представляется рациональным. Например, CustTbl, представляющий таблицу клиентов, используется исключительно для вывода полного списка клиентов и для манипуляций над ним же. В то же время, компонент BillTbl было бы рационально заменить на компонент-запрос, поскольку по счетам будет производиться выборка. Кроме того, если рассматривать приближенный к практике случай, то таблица счетов со временем может стать весьма и весьма объемной, что в случае работы по сети может сказаться на производительности. Поэтому заменим этот компонент запросом Query и назовем его BillQry. При этом свойство DatabaseName у него так же будет MainDB. Кроме того, поскольку этот запрос у нас используется в качестве источника данных для таблицы DBGrid, в которую мы позволяем вносить правку, то свойство RecuestLive следует установить в истину. С учетом того, что в данном запросе используется только одна таблица (bill), это позволит нам обращаться с ним

так же легко, как с обычной таблицей. После этого останется изменить в BillDS значение свойства DataSet – вместо ссылки на отсутствующий теперь компонент BillTbl укажем для него BillQry.

Наконец, нам понадобится еще один компонент Query – для выполнения вспомогательных задач – например, для добавления новых записей в таблицы. Назовем его StdQry и установим свойство DatabaseName в MainDB. Поскольку этот запрос не будет непосредственно выводить информацию, то добавляеть ему в пару компонент DataSource не требуется.

Теперь приступим к изменению в программном коде. Начнем с функции вывода баланса – обработчика события Click для кнопки BalanceBtn. Как раз в этом случае мы задействуем оба запроса – один для того, чтобы отобразить все нужные счета (BillQry), а другой – для того, чтобы вычислить сумму (StdQry). Вариант этой функции с использованием таблиц был приведен в листинге 20.4, теперь же мы рассмотрим новый вариант, с использованием SQL-запросов (листинг 21.2).

Листинг 21.2. Вывод счетов и вычисление суммы с использованием SQL

```
procedure TMainFrm.BalanceBtnClick(Sender: TObject);
begin
 Data.BillQry.Close;
 Data.BillQry.SQL.Text:='SELECT * FROM bill WHERE BILL_CUST='
+Data.CustTbl.FieldByName('CUST_ID').AsString;
 Data.BillQry.Open;
 Data.StdQry.Close;
 Data.StdQry.SQL.Text:='SELECT SUM(BILL_SUMM) AS TOTAL FROM bill WHERE '
+'BILL_CUST='+Data.CustTbl.FieldByName('CUST_ID').AsString;;
 Data.StdQry.Open;
 BillsFrm.Caption:=Data.CustTbl.FieldByName('CUST_NAME').AsString+': '
+Data.StdQry.FieldByName('TOTAL').AsString;
 BillsFrm.ShowModal;
end;
```

Очевидно, что вычислительные задачи в новом варианте, при помощи SQL мы полностью переложили на саму СУБД. Кроме того, объем информации, полученный приложением, заметно уменьшился, поскольку выборка нужных значений производится на стороне СУБД, и приложению будут переданы только те данные, которые реально необходимы.

Теперь перепишем код для кнопок «Новый счет» и «Новый клиент». В данном случае, поскольку производить выборки не требуется, нам будет достаточно только одного запроса – StdQry, как это показано в листинге 21.3.

Листинг 21.3. Создание новых записей в таблицах bill и customer при помощи SQL

```
procedure TMainFrm.NewBillBtnClick(Sender: TObject);
begin
 with BillFrm do begin
   Caption:='Счет для '+Data.CustTbl.FieldByName('CUST_NAME').AsString;
   ShowModal;
   if ModalResult<>mrOk then exit;
   Data.StdQry.Close;
```

```
   Data.StdQry.SQL.Text:='INSERT INTO bill(BILL_CUST, BILL_SUMM, BILL_DATE)'
+' VALUES ('+Data.CustTbl.FieldByName('CUST_ID').AsString+', '+
   SummEd.Text+', '''+DateToStr(DatePick.Date)+''')';
   Data.StdQry.ExecSQL;
 end;
end;
procedure TMainFrm.NewCustBtnClick(Sender: TObject);
begin
 with CustFrm do begin
   ShowModal;
   if ModalResult<>mrOk then exit;
   Data.StdQry.Close;
   Data.StdQry.SQL.Text:='INSERT INTO customer (CUST_NAME, CUST_ADDRESS) '
+'VALUES ('''+NameEd.Text+''', '''+AddrEd.Text+''')';
   Data.StdQry.ExecSQL;
   Data.CustTbl.Refresh;
 end;
end;
```

Здесь следует отметить следующий момент: после выполнения запроса на добавление клиента для компонента, представляющего таблицу клиентов (CustTbl), вызывается метод Refresh. Если этого не сделать, то, хотя данные и будут добавлены в таблицу физически, обновления таблицы на экране не произойдет. В то же время для добавления счетов подобной операции делать не требуется, поскольку для того, чтобы просмотреть список счетов, пользователю в любом случае придется нажимать на кнопку «Баланс», процедура обработки которой будет всякий раз составлять и вызывать новый запрос.

Что касается изменения учетной записи клиента, то в данном случае, в принципе, можно ничего не делать, поскольку мы оставили таблицу. С другой стороны, вполне можно использовать для этих целей язык SQL и компонент запроса StdQry. Для этого достаточно заменить последние 4 строчки кода в обработчике двойного щелчка мышью у компонента CustGrd:

```
//старый вариант
   Data.CustTbl.Edit;
   Data.CustTbl.FieldByName('CUST_NAME').AsString:=NameEd.Text;
   Data.CustTbl.FieldByName('CUST_ADDRESS').AsString:=AddrEd.Text;
   Data.CustTbl.Post;
//новый вариант
   Data.StdQry.Close;
   Data.StdQry.SQL.Text:='UPDATE customer SET CUST_NAME='''+NameEd.Text
+''', CUST_ADDRESS='''+AddrEd.Text+'''';
   Data.StdQry.ExecSQL;
```

В то же время, подобный код в форме BillsFrm, несмотря на то, что в данном случае источник данных изменился с таблицы на запрос, можно не менять – как раз в этом случае и пригодится установленное в истину свойство запроса RequestAlive.

С полным кодом переработанного приложения можно ознакомиться в каталоге *Demo\Part4\DBAppSQL*.

Отчеты

Вопрос рассмотрения работы с базами данных в Delphi был бы не полным, если не рассмотреть работу с отчетами. Отчеты представляют собой печатные документы, получаемые в результате выполнения запросов к БД. А распечатка всевозможных бумажных документов – одно из основных предназначений приложений БД.

Отчеты в Delphi

Для упрощения процесса составления отчетов в поставку всех версий Delphi входят специальные программные средства – генераторы отчетов. В первых версиях Delphi это был ReportSmith, затем, с Delphi 3 по Delphi 6 – QuickReport, а начиная с Delphi 7 и заканчивая Delphi 2006 – Rave Reports.

ПРИМЕЧАНИЕ

> В поставку Delphi 7 так же входит и QuickReport, однако по умолчанию этот инструментарий отсутствует на панели компонентов. Чтобы воспользоваться QuickReport в этой версии Delphi, следует открыть Component → Install Packages, нажать кнопку Add и выбрать файл dclqrt70.bpl в подкаталоге …Delphi\Bin.

Генераторы ReportSmith и Rave Reports представляют собой отдельные приложения, при помощи которых можно создавать отчеты, в то время, как QuickReports – это набор VCL-компонентов, которые помещают непосредственно на стандартную форму Delphi.

В простейшем случае работа с QuickReport выглядит следующим образом: создается новая форма, на которую помещают компонент QuickRep. Затем при помощи составного свойства Bands определяют, какие основные составные части отчета требуются:

- HasColumnHeader – заголовки столбцов;
- HasDetail – сами столбцы (собственно данные);
- HasPageFooter – нижний колонтитул страницы;
- HasPageHeader – верхний колонтитул страницы;
- HasSummary – итоговое значение по столбцам данных;
- HasTitle – общий заголовок отчета.

В типичном случае, как минимум, необходимы столбцы и их заголовки. Общий заголовок так же обычно необходим. Кроме того, для потенциально многостраничных отчетов, как правило, предусматривают нумерацию страниц, для чего пригодится тот или иной колонтитул.

Рассмотрим простейшее приложение, использующее отчет. Для этого создадим новый проект в Delphi (если у вас Delphi 7, то сначала подключите QuickReport, как описано в примечании, если же у вас более новая версия – то можете пропустить этот текст и сразу перейти к следующему параграфу). Затем на главную форму поместите 3 кнопки – «Просмотр», «Печать» и «Выход», а в заголовке окна (Caption) напишите «QuickReport demo» (рис. 22.1).

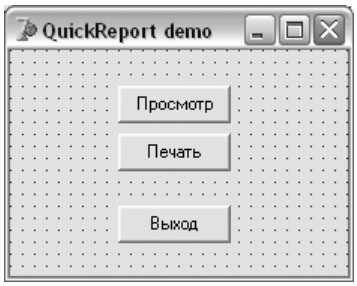

Рис. 22.1. Приложение QuickReport demo

Теперь создадим новую форму, поместим на нее компонент Table, установим свойство DatabaseName в DATA1, а для TableName выберем Customer, после чего установим свойство Active в истину. Таким образом, мы подготовили данные для вывода. Теперь поместим на форму компонент QuickRep, который визуально предствляет собой лист отчета и установим в свойстве Bands флаги HasColumnHeader, HasDetail, HasPageFooter и HasTitle. В результате на «листе» появятся 4 области, представляющие собой соответствующие логические части отчета. Теперь в центре самой верхней области – Title – поместим компонент QRLabel. Это аналог стандартной подписи (Label), предназначенный специально для отчетов QuickReport. В свойстве Caption напишем «Клиенты», после чего можно увеличить размер шрифта, сделать его полужирным или выбрать другую гарнитуру, чтобы заголовок не «затерялся».

Следующим этапом будет собственно подготовка к выводу данных отчета. Поскольку мы выбрали таблицу клиентов, то выводить будем 2 колонки – номер клиента в базе (поле CUST_ID) и иго имя (CUST_NAME). Для этого сначала на области заголовков столбцов (Column Header) разместим еще 2 надписи, воспользовавшись QRLabel, в свойстве Caption которых напишем «№» и «Имя». В следующей области, Detail, непосредственно под этими надписями, поместим 2 компонента QRDBText. Эти компоненты, в свою очередь, можно назвать аналогами компонент DBText, предназначенных для вывода данных из БД в строковом формате. Свойство DataSet компонентов QRDBText следует установить в Table1, а DataField – в CUST_ID у одного и в CUST_NAME – у другого. Здесь надо отметить, что компоненты, помещенные в области Detail, при печати будут повторяться построчно столько раз, сколько требуется для вывода всех данных из указанного источника. Но следует учитывать, что и у компонентов, используемых для вывода полей, и у основы отчета – компонента QuickRep, должен использоваться один и тот же источник данных. В данном случае это Table1.

Последнее, что остается – это определить содержимое нижнего колонтитула. Для этого в области Page Footer поместим компонент QRSysData. Прямого аналога в VCL у этого компонента нет, можно назвать его «продвинутой» версией текстовой подписи, которая может менять свое содержимое в зависимости от состояния отчета. За тип выводимой информации отвечает свойство Data, которое может принимать одно из следующих значений: qrsDare, qrsDateTime, qrsDetailCount, qrsDetailNo, qrsPageNumber, qrsReportTitle и qrsTime. Соответственно, будет выводиться текущая дата, дата и время, количество записей на странице, номер последней записи на странице, номер страницы, название отчета, или текущее время. Поскольку записей в таблице у нас весьма немного, то остановимся на втором варианте – выводе даты и времени состав-

ления отчета. Впрочем, при необходимости всегда можно использовать несколько таких компонентов, чтобы вывести всю необходимую информацию.

В результате, после всех действий по настройке отчета, мы получим заготовку, представляющую собой изображение листа с заданными нами подписями и намеченными местами для вывода данных (рис. 22.2).

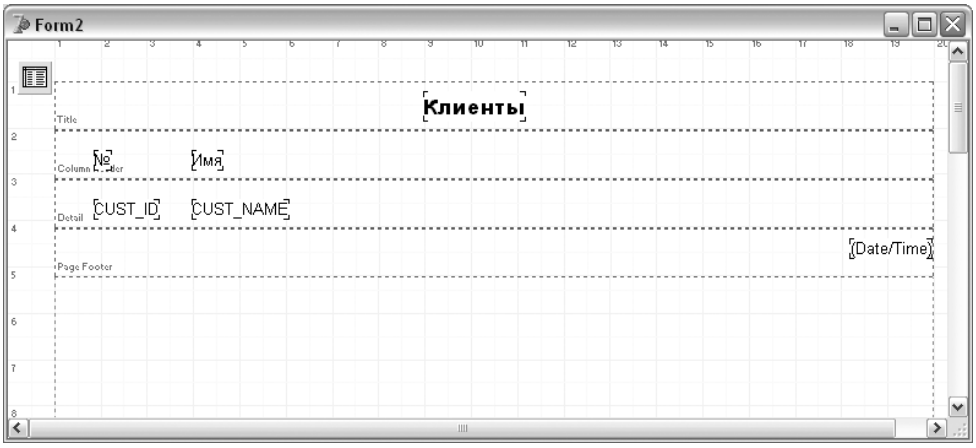

Рис. 22.2. Форма Delphi с отчетом QuickReport

Теперь остается написать код для обращения к этому отчету. Для этого вернемся к первой (главной) форме приложения, в начале секции implementation подключим форму с запросом при помощи ключевого слова uses и создадим обработчики событий для кнопок, как показано в листинге 22.1.

Листинг 22.1. Исходный код приложения QuickReport demo

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, QRCtrls, QuickRpt, ExtCtrls, DB, DBTables;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button3Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  end;
var
  Form1: TForm1;
implementation
uses unit2;
{$R *.dfm}
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
 close;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
 Form2.QuickRep1.Preview;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
 Form2.QuickRep1.Print;
end;
end.
```

Как видно из кода, вся программная работа заключается в вызове метода Preview для просмотра отчета и метода Print для вывода на печать. С исходным кодом так же можно ознакомится в каталоге *Demo\Part4\QReport*.

Основы Rave Reports

Начиная с Delphi 7, фирма Borland вернулась к идее использования отдельного приложения для создания отчетов. На сей раз, вместо генератора ReportSmith, было использовано другое приложение – Rave Reports. Одним из ключевых преимуществ этого генератора отчетов является то, что его ядро можно включить в состав собственного приложения. Таким образом, не требуется явно устанавливать библиотеки Rave Reports на все ПК, на которых будет выполняться разрабатываемое вами приложение БД.

В зависимости от того, какая версия Delphi у вас установлена – 7.0, 2005 или 2006, в вашем распоряжении окажется Rave Reports версии 5.0, 6.0 или 6.5. Существенных различий между ними нет, поэтому мы будем рассматривать версию 5.0. При необходимости, Rave Reports можно использовать и с более ранними версиями Delphi, а так же с рядом других IDE.

Что касается самого генератора Rave Reports, то он состоит из 3 частей:

- Ядра, которое обеспечивает управление отчетом, его предварительный просмотр и отправку на печать. Исполняемый код ядра включается в приложение Delphi при компиляции, делая его полностью автономным при работе на компьютере клиента;
- Визуальная среда разработки отчетов, предназначенная для разработки самих отчетов. Она позволяет добавлять к отчету страницы, размешать на них графические и текстовые элементы, подключать к отчетам источники данных и т.д. Отчеты сохраняются в файлах с расширением RAV и должны распространяться вместе с использующими их приложениями;
- Компоненты Rave Reports, обеспечивающие управление отчетами из приложения. Они расположенные на странице Rave палитры компонентов.

Основным преимуществом Rave Reports перед QuickReport является собственная визуальная среда разработки отчетов, которая, в принципе, позволяет добиться

больших результатов при меньших усилиях. Правда, обратной стороной этого улучшения является то, что предварительно придется изучить эту среду, что несколько затруднительно, особенно учитывая весьма скупую справочную систему, поставляемую вместе с этим генератором отчетов. Поэтому для начала остановимся на самой интегрированной среде разработки Rave Reports, тем более что разработка отчета при помощи Rave начинается как раз с его конструирования в самой среде, и лишь после этого в разрабатываемом приложении в Delphi настраивается связь приложения с отчетом при помощи компонентов, расположенных на странице Rave.

Среда Rave Reports

Вызвать визуальный конструктор Rave Reports можно из меню Delphi, выбрав пункт Tools→Rave Designer. В результате запустится приложение – среда разработки отчетов Rave Reports (рис. 22.3).

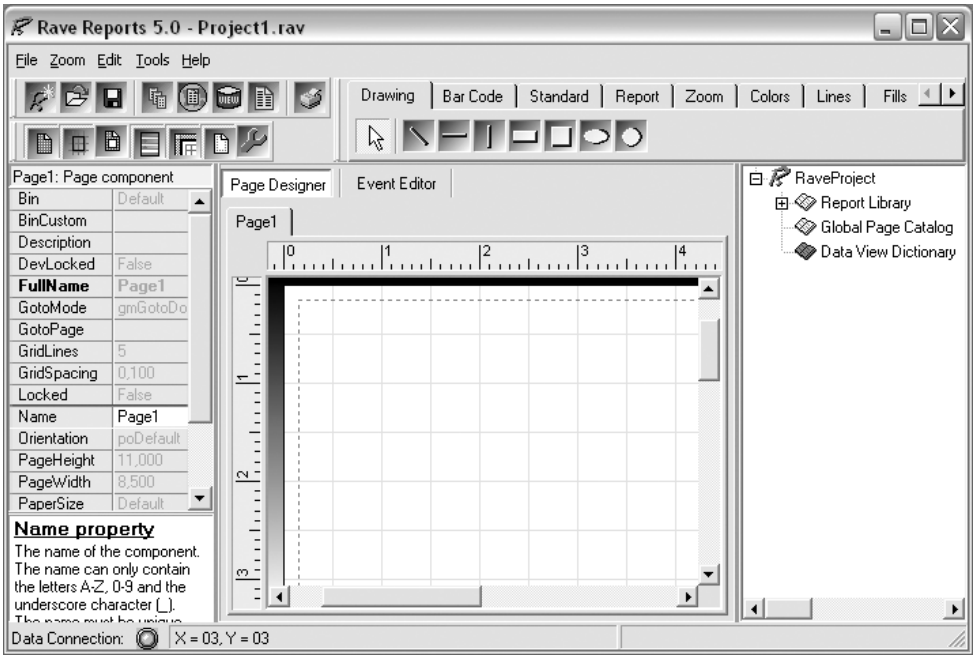

Рис. 22.3. Вид окна IDE Rave Reports 5.0

Интерфейс визуального конструктора отчетов вполне стандартен для Windows-приложений и напоминает интерфейс самой среды Delphi. В самом верху расположено меню, а под ним – панели инструментов. Слева расположены 2 панели, дублирующие наиболее востребованные команды программы. Справа располагается многостраничная панель инструментов, подобная палитре компонентов в Delphi, на ней находятся компоненты, при помощи которых и создаются отчеты.

Ниже расположена основная часть окна, разделенная на 3 части. Так, слева расположен инспектор компонентов отчета, при помощи которого настраиваются свойства выбранного компонента, подобно тому, как это осуществляется в инспекторе объекта в IDE Delphi.

В центральной части окна визуального конструктора располагается блокнот с 2 вкладками – Page Designer (конструктор страниц) и Event Editor (редактор событий). На вкладке конструктора страниц можно добавлять, удалять и редактировать компоненты отчета. При этом, поскольку отчеты могут быть многостраничными, то эта вкладка, в свою очередь, так же является блокнотом, каждый лист которого представляет собой страницу отчета. Что касается вкладки редактора событий, то на ней, при необходимости, определяют обработчики событий для отчетов и их составных частей – страниц, элементов оформления и т.д.

Справа расположено дерево проекта отчета, которое отображает состав проекта и в котором можно выбрать нужный компонент. При помощи двойного щелчка мышкой по выбранному в дереве элементу можно выделить соответствующий элемент на странице и увидеть его свойства в инспекторе компонентов отчета

Само дерево состоит из 3 «ветвей»:

- Report Library – библиотека отчетов, содержащая все отчеты проекта вместе со всеми их составляющими – страницами, элементами оформления и т.д., упорядоченные в иерархической структуре;
- Global Page Catalog – каталог глобальных страниц, содержащий страницы, общие для всех отчетов проекта;;
- Data View Dictionary – словарь просмотров данных, содержащий объекты соединения с данными из внешних источников.

Здесь следует отметить, что все создаваемые в Rave Reports отчеты являются частью проектов. Проект может содержать произвольное число отчетов, а каждый отчет, в свою очередь, может состоять из произвольного числа страниц. При этом словарь просмотров данных, как и страницы из глобального каталога являются общими для всех отчетов проекта. При работе с проектом выделяют какой-либо отчет, который называется текущим отчетом. Сразу после открытия проекта текущим становится первый отчет, но при необходимости всегда можно выбрать нужный отчет при помощи двойного щелчка мышкой по его названию в дереве проекта.

Чтобы добавить новый отчет к проекту с помощью команды New Report из меню File, а новая страница в отчете добавляется при помощи команды New Report Page из этого же меню. Сразу после создания объект незамедлительно появляется в соответствующей ветви дерева проекта.

Если проект включает в себя несколько отчетов, и требуется создать страницу, которая будет использоваться в нескольких их них, то следует создать страницу для каталога глобальных страниц при помощи команды New Global Page. После создания такую страницу можно редактировать и добавлять в состав любого входящего в проект отчета.

Состав и порядок вывода страниц отчета определяется при помощи свойства PageList отчета. Это свойство доступно из инспектора компонента отчета, когда в дереве проекта выбран сам отчет. Для выбора страниц и их упорядочивания следует нажать на кнопку с многоточием, при этом появится редактор страниц отчета (Page List Editor), при помощи которого определяют выводимые на печать страницы, а так же настраивают порядок их вывода (рис. 22.4).

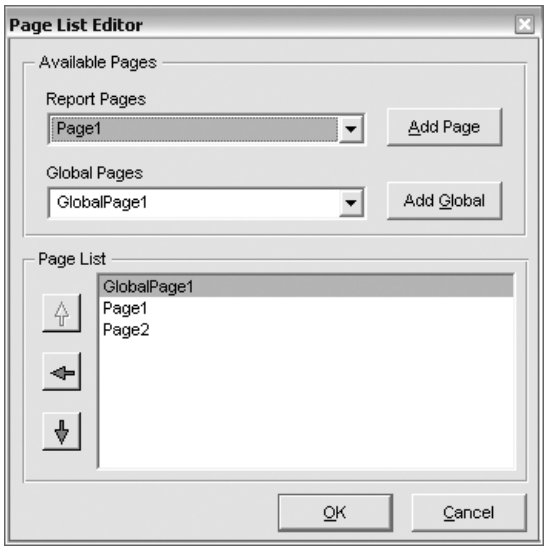

Рис. 22.4. Редактор страниц отчета

Все доступные страницы отчета находятся в 2 списках. В первом (Report Pages) доступны все страницы текущего отчета, а во втором (Global Pages) – все глобальные страницы проекта. При нажатии кнопок Add Page и Add Global страницы попадают в список Page List. Уже после внесения в список можно менять порядок вывода страниц, а так же удалять их из этого списка при помощи кнопок, расположенных слева.

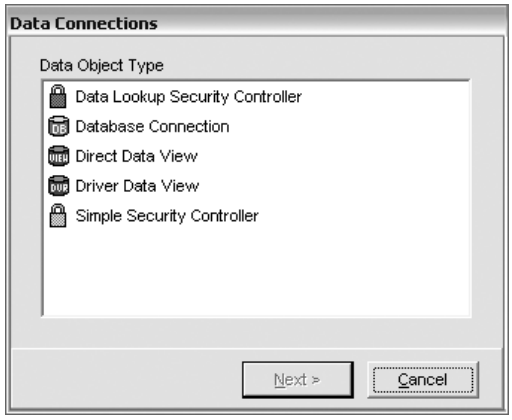

Рис. 22.5. Создания объекта данных – выбор типа источника

Наконец, новый источник данных создается при помощи команды New Data Object. При этом открывается диалоговое окно Data Connections (рис. 22.5), в котором следует выбрать один из 5 предлагаемых типов объектов:

- Data Lookup Security Controller – контроллер безопасности поиска данных, обеспечивающий авторизацию пользователей по имени и паролю;

- Database Connection – соединение с БД, устанавливает соединение с внешним источником данных;
- Direct Data View – прямой просмотр данных, создает просмотр данных для активного соединения с источником данных;
- Driver Data View – просмотр данных через драйвер, создает просмотр данных на основе уже имеющегося соединения;
- Simple Security Controller – простой контроллер безопасности, определяет список пользователей для организации доступа.

После выбора типа, если выбран вариант соединения с базой данных, будет предложено определиться с типом источника – BDE, ADO или dbExpress (DBX), а затем, в зависимости от выбранного варианта, будет предложено либо указать псевдоним БД в BDE, либо определить параметры соединения.

ПРИМЕЧАНИЕ

> В Rave Reports 6.x имеется так же поддержка источников данных IBX – Interbase Express. Так же можно отметить, что версии 6 и 6.5 выгодно отличаются от более ранних наличием полноценной справочной системы.

После того, как источник данных создан, он появляется в дереве проекта отчетов в группе Data View Dictionary. Все источники данных являются общими и доступны для всех отчетов проекта.

Компоненты Rave Reports для создания отчетов

Для создания и оформления отчетов в среде Rave Reports имеется многостраничная панель инструментов, имеющая 10 вкладок:

- Drawing – графические компоненты для оформления отчета;
- Bar Code – компоненты для вывода различных вариантов штрихкода;
- Standard – компоненты для отображения текста и графики;
- Report – компоненты для отображения данных;
- Zoom – компоненты для изменения масштаба отображения при разработке отчета;
- Colors – компоненты для выбора цвета графических элементов оформления;
- Lines – компоненты для определения толщины и стиля линий графических элементов оформления;
- Fills – компоненты для определения типа заливки элементов оформления;
- Fonts – компоненты для определения параметров текста;
- Alignment – компоненты для выравнивания элементов отчета на странице.

По сути, именно компонентами являются только те элементы, что расположены на первых 4 вкладках – Drawing, Bar Code, Standard и Report. При этом компоненты,

расположенные во вкладках Drawing, Bar Code и Standard используются для визуального оформления и вывода статичной информации, в то время как собственно за вывод данных отвечают компоненты, расположенные на вкладке Report. Названия и описания всех компонентов, предназначенных для оформления отчетов, сведены в таблице 22.1.

Таблица 22.1. Компоненты для оформления отчетов в Rave Reports

Компонент	Вкладка	Описание
Line, Hline, Vline	Drawing	Компоненты для рисования универсальной, горизонтальной и вертикальной линии. Разница между универсальной линией и остальными заключается в том, что ее можно «повернуть» при помощи мышки, а другие – только путем редактирования свойства Height в инспекторе компонентов отчета
Rectangle, Square	Drawing	Компоненты для вывода прямоугольника и квадрата
Circle, Ellipse	Drawing	Компоненты для вывода окружности и эллипса
PostNetBarCode	Bar Code	Компонент для вывода штрихкода в формате PostNet, который применяется в почтовой службе США
I2of5BarCode	Bar Code	Компонент для вывода штрихкода в формате I2of5, предназначенного для вывода цифровых последовательностей
Code39BarCode	Bar Code	Компонент для вывода штрихкода в формате Code39, кодирующего цифры и заглавные латинские буквы, а так же некоторые иные символы
Code128BarCode	Bar Code	Компонент для вывода штрихкода в формате Code128, при помощи которого можно представить первую половину символов таблицы ASCII
UPCBarCode	Bar Code	Компонент для вывода штрихкода в формате UPC, служащего для маркировки товаров 12-ю цифрами
EANBarCode	Bar Code	Компонент для вывода штрихкода в формате EAN, служащего для маркировки товаров 13-ю цифрами
Text	Standard	Компонент для вывода однострочного текста. Сам текст задается при помощи свойства Text путем его редактирования в инспекторе компонентов отчета
Memo	Standard	Компонент для вывода многострочного текста. Сам текст задается при помощи свойства Text
Section	Standard	Непечатный компонент, предназначенный для группировки нескольких компонентов
Bitmap	Standard	Компонент для отображения растровых изображений в формате BMP. Само изображение задается при помощи свойства Image
MetaFile	Standard	Компонент для отображения векторных изображений в форматах EMF и WMF. Само изображение задается при помощи свойства Image
FontMaster	Standard	Непечатный компонент, предназначенный для определения свойств шрифта для компонентов отчета, связанных с данным элементом при помощи свойства FontMirror
PageNumInit	Standard	Непечатный компонент, предназначенный для переопределения нумерации страниц начиная с той страницы отчета, на которой он расположен

Компоненты для вывода данных, в свою очередь, подразделяются на следующие группы: компоненты отображения данных, обеспечивающие вывод данных из источника; структурные компоненты, предназначенные для группировки; компоненты вычислений, предназначенные для произведения математических операций над выводимыми в отчете данными. Все они приведены в таблице 22.2.

Таблица 22.2. Компоненты вывода данных в Rave Reports

Компонент	Группа	Описание
DataText	Отображение	Используется для вывода строковых и числовых значений из связанного с ним источника (просмотра) данных
DataMemo	Отображение	Используется для вывода данных в формате Memo или BLOB
CalcText	Отображение	Используется для вывода результатов вычислений по связанным полям
DataMirrorSection	Отображение	Аналог компонента Section, предназначенный для компонентов вывода данных
Region	Структура	Используется для определения области (части страницы), на которой располагаются другие компоненты вывода данных. При необходимости можно разбить область на колонки при помощи свойства Columns
Band	Структура	Используется для определения полосы, на которой располагаются компоненты оформления отчета. Компонент Band может размещаться только на компоненте Region
DataBand	Структура	Используется для выделения полосы, на которой располагаются компоненты вывода данных. Компонент DataBand может размещаться только на компоненте Region
DataCycle, CalcOp, CalcTotal, CalcController	Вычисления	Группа непечатных компонентов, используемых для вычисления данных на основе информации, доступной из источников (просмотров) данных

Что касается всех остальных компонентов, то они, по сути, являются лишь командами для изменения свойств отображения элементов отчета.

Для добавления компонента к отчету достаточно выбрать его на панели инструментов и щелкнуть по нужному листу отчета. При этом печатные компоненты появляются как на листе – в том месте, где был произведен щелчок мышкой, так и в дереве проекта. Непечатные же компоненты на листах не отображаются – доступ к ним возможен только из дерева проекта отчета.

Компоненты Rave Reports в VCL

Для взаимодействия с приложениями, создаваемыми в Delphi, вместе с Rave Reports поставляется набор VCL-компонент. В Delphi 7 и более поздних версиях они установлены по умолчанию и находятся на вкладке Rave палитры компонентов Delphi.

Все компоненты Rave можно условно разделить на 4 категории – для управления проектом, для организации соединений с данными, для вывода на печать или на экран, а так же для экспорта в какой-либо формат. Всего же их насчитывается 13:

- RvProject – ключевой компонент, используемый для взаимодействия между отчетом Rave и приложением Delphi;

- RvSystem – компонент, обеспечивающий взаимодействие между отчетом и функциями по печати и просмотру отчета. Фактически, RvSystem является интегрированной версией RvNDRWriter, RvRenderPreview и RvRenderPrinter;
- RvCustomConnection – обеспечивает соединение с источником данных типа текстового файла, таблицы Excel и т.п.;
- RvDataSetConnection – обеспечивает соединение с набором данных СУБД;
- RvTableConnection – обеспечивает соединение с компонентом типа Table;
- RvQueryConnection – обеспечивает соединение с компонентом типа Query;
- RvNDRWriter – компонент для подготовки отчета в специальном двоичном формате для дальнейшего вывода на принтер или для просмотра на экране;
- RvRenderPreview – компонент, обеспечивающий вывод отчета на экран;
- RvRenderPrinter – компонент, обеспечивающий вывод отчета на печать;
- RvRenderPDF, RvRenderHTML, RvRenderRTF, RvRenderText – группа компонентов, обеспечивающих экспорт отчета в форматы PDF, HTML, RTF или в текстовый файл.

Наиболее важным для построения отчета компонентом, безусловно, является компонент RvProject, поскольку без его участия невозможно взаимодействие приложения с отчетами Rave Reports. Связь с проектом отчета осуществляется при помощи свойства Project File, которому присваивается имя RAV-файла с проектом отчета:

```
RvProject1.ProjectFile:='c:\project1.rav';
```

Поскольку в одном проекте Rave Reports может находиться несколько отчетов, то для выбора нужного отчета используется свойства ReportName, ReportFillName и ReportDesc, позволяющие сослаться на нужный отчет по его короткому или полному имени, или по описанию. Чтобы получить список всех отчетов проекта, можно прибегнуть к процедуре GetReportList, которая возвращает список всех отчетов проекта. При этом в качестве первого параметра, передаваемого этой процедуре, должен быть передан объект типа TStrings, а в качестве второго – булево значение, в зависимости от которого список будет заполнен либо сокращенными, либо полными названиями отчетов.

С другими компонентами Rave, используемыми для управления отчетом, компонент RvProject связывается при помощи свойства Engine. В типичном случае, в качестве Engine указывают компонент RvSystem, при помощи которого можно выводить отчет как на экран, так и на печать, или же экспортировать его в файл. Фактически, работа с отчетом Rave Reports в Delphi сводится к тому, что сначала при помощи компонента RvProject устанавливается связь с проектом и выбирается нужный отчет, после чего при помощи RvSystem отчет выводится на печать, в файл, или на экран.

Компоненты RvCustomConnection, RvDataSetConnection, RvTableConnection и RvQueryConnection предназначены для предоставления отчету источников данных из приложения Delphi. Именно эти компоненты будут отображаться при разработке отчета в среде Rave Reports, если при создании объекта данных (New Data Object) выбрать вариант прямого просмотра данных (Direct Data View). Разумеется, при этом должна

быть запущена среда Delphi с загруженным приложением, содержащим подобные компоненты.

Пример создания отчета

Рассмотрим пример создания простейшего отчета с использованием Rave Reports. Пусть он будет выводить ту же самую таблицу и примерно в таком же виде, как и рассмотренный в начале этой главы отчет, созданный на базе VCL-компонентов QuickReport.

Для начала нам потребуется главная форма приложения, подобная той, что была использована в примере с QuickReport (см. рис. 22.1). Этой формой и ограничимся, поскольку в данном случае форма отчета создается не средствами VCL-компонент в IDE Delphi, а в специализированной среде Rave Reports. Поэтому единственный компонент доступа к данным – Table – мы так же поместим на этой форме. При этом, как и в предыдущем случае, установим свойство DatabaseName в DATA1, TableName – в customer, а Active – в истину.

Теперь перейдем к компонентам, расположенным на закладке Rave. Из них нам понадобятся RvProject – основной компонент отчета, RvSystem – для генерации отчетов и вывода их на принтер или в окно просмотра, а так же RvDataSetConnection – для взаимодействия с источником данных, т.е. с компонентом Table. Соответственно, следует связать компонент RvDataSetConnection с Table, установив у него в свойстве DataSet значение Table1. Что касается компонента RvProject, то его следует связать с компонентом RvSystem, для чего в свойстве Engine следует установить значение RvSystem1.

После этого можно приступить собственно к проектированию отчета, для чего нам понадобится загрузить среду Rave Reports. Сделать это можно как из меню Tools, так и при помощи двойного щелчка мышкой по компоненту RvProject.

В среде Rave Reports следует начать новый проект, выбрав пункт New из меню File. При этом автоматически создастся новый проект, уже содержащий пустой одностраничный отчет. Для начала подготовим заголовок страницы отчета, для чего вверху страницы, по центру, расположим текстовую надпись (компонент Text расположен на вкладке Standard), в свойстве Text которой, при помощи инспектора компонентов, укажем значение «Клиенты». Чтобы выделить эту надпись шрифтом, можно перейти на закладку Fonts и настроить вид надписи при помощи имеющихся на ней элементов управления. Затем наметим заголовки столбцов, для чего несколько ниже основного заголовка разместим еще 2 компонента Text, один из которых будет являться заголовкам для колонки с номерами клиентов в базе, а другой – для имен.

Таким образом, можно приступать к подготовке к выводу собственно данных. Но, прежде всего, нам потребуется определить источник данных, для чего следует создать объект данных (File → New Data Object). В окне диалога Data Connections следует выбрать вариант Direct Data View и нажать на кнопку Next, после чего появится второй экран этого диалога со списком доступных соединений. Впрочем, в данном случае, список будет состоять всего из 1 элемента – RvDataSetConnection1, который мы и выберем. После нажатия на кнопку Finish, новый источник данных появится в дереве проекта под именем DataView1.

После всей проделанной подготовительной работы займемся выводом данных, для чего нам понадобятся компоненты отчета, расположенные на закладке Report. Прежде всего, поместим на лист компонент Region и при помощи мышки растянем его таким образом, чтобы он занял практически все свободное место на листе, поскольку именно размеры этого компонента определяют область, в которую будут выводиться данные. Затем, непосредственно на этот компонент, поместим другой – DataBand. На компонент DataBand, в свою очередь, поместим 2 компонента DataText, выровняв их по левому краю с заголовками столбцов (рис. 22.6).

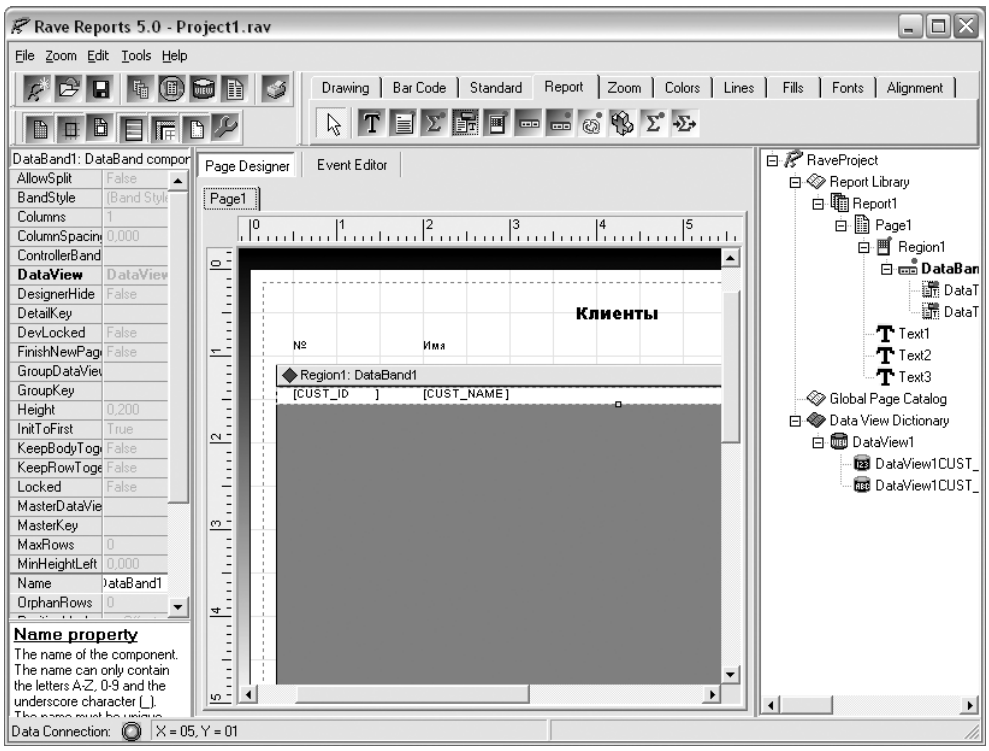

Рис. 22.6. Разработанный отчет в среде Rave Reports

Теперь остается связать Data-компоненты с источником данных, для чего как для компонента DataBand, так и для обоих компонентов DataText следует установить свойство DataView в DataView1. Для компонентов DataText помимо этого следует выбрать еще и поля, которые они будут выводить, что определяется при помощи свойства DataField. Соответственно для левого компонента это будет CUST_ID, а для правого – CUST_NAME.

Фактически, разработку отчета можно считать завершенной, и можно сохранить проект, причем желательно в том же каталоге, что и приложение Delphi, которое данный отчет использует. После сохранения можно закрыть среду Rave Reports и вернуться к разработке в Delphi.

Прежде всего, нам следует указать имя файла проекта в свойстве ProjectFile компонента RvProject. В том случае, если проект был сохранен в том же каталоге, что и это приложение, то достаточно просто ввести имя файла. В противном случае потребует-

ся указать полный путь, для чего можно нажать кнопку с многоточием напротив этого поля в инспекторе объектов и воспользоваться стандартным диалогом открытия файла. Кроме того, поскольку с одной стороны, при выполнении запроса на вывод отчета компонент RvSystem запрашивает, куда его выводить, а с другой у нас и так имеется 2 отдельных кнопки – для вывода на экран и на принтер, то в свойстве SystemSetups этого компонента следует выключить флаг ssAllowSetup.

Вся подготовительная работа, связанная с созданием отчета и настройке компонент на этом заканчивается и можно приступать к написанию программного кода, которого, собственно, будет весьма немного. В частности, нам надо лишь указать на то, куда выводить отчет и собственно обеспечить вызов процедуры вывода. Делается это 2 строками кода:

```
RvSystem1.DefaultDest:=rdPreview;
RvProject1.Execute;
```

Здесь в первой строке указывается, что вывод следует производить в окно просмотра на экран монитора, а во второй обеспечивается сам вывод. Для вывода же на печать следует лишь изменить значение, назначаемое свойств DefaultDest с rdPreview на rdPrinter:

```
RvSystem1.DefaultDest:=rdPrinter;
RvProject1.Execute;
```

Вместе с тем, следует учитывать, что для вывода из проекта будет браться первый попавшийся отчет, поэтому если отчетов будет несколько, то всякий раз следует указывать, какой именно отчет нам требуется. Сделать это можно при помощи метода SelectReport:

```
RvProject1.SelectReport('Report1',true);
```

Кроме того, не было бы лишним явно загрузить отчет, для чего можно воспользоваться методом Open:

```
RvProject1.Open;
```

В результате, мы получим код, приведенный в листинге 22.2.

Листинг 22.2. Код приложения Rave Reports demo

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 RvProject1.SelectReport('Report1',true);
 RvProject1.Open;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
 RvSystem1.DefaultDest:=rdPreview;
 RvProject1.Execute;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
RvSystem1.DefaultDest:=rdPrinter;
 RvProject1.Execute;
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
 close;
end;
```

С самим приложением можно ознакомиться в каталоге *Demo\Part4\RReport*.

Вместо заключения

Если книга вам понравилась и существенно помогла в изучении Delphi, то вы можете отблагодарить автора, например, отправив любую сумму в качестве гонорара через платежную систему WebMoney на один из следующих счетов:

Z344236556433
R927736520300
E317486821433

Так же вы можете связаться со мной через сайт www.snkey.net или по электронной почте serge@snkey.com. Буду рад так же, если вы сообщите о найденных неточностях, к счастью электронный формат издания позволяет безболезненно вносить в книгу коррективы.

Содержание

Предисловие к электронной редакции **2**

Введение **3**

ЧАСТЬ I. ОСНОВЫ ПРОГРАММИРОВАНИЯ В DELPHI 5

О программировании и о Delphi **5**
Что такое программирование 5
Что такое Delphi 5
Delphi и другие 6
Процедурное программирование и алгоритмы 7
Об объектно-ориентированном программировании 8
Визуальное программирование и Delphi 9
Версии Delphi и их отличия 10

Среда Delphi и простейшее приложение **11**
Состав Delphi и требования к системе 12
Интегрированная среда разработки 13
Проекты в Delphi 18
Типы проектов и депозитарий 22
Прочие средства IDE 23
Создание приложения командной строки 25

Введение в Object Pascal **27**
Алфавит и словарь языка 27
Ключевые слова 28
Переменные и константы 29
Типы данных 31
Данные и значения 34
Операции и их типы 35
Выражения и приоритет операций 38
Структура программы 39

Операторы Object Pascal **41**
Об управляющих структурах 41
Условный оператор if 42
Оператор выбора case 43
Оператор цикла for 45
Операторы циклов while и repeat 46
Операторы break и continue 47
Операторы в действии 48

Структурные типы данных **51**
Пользовательские типы данных 51
Массивы 51

Множества 54
Записи....56
Специальные типы данных 59
Совместимость и преобразование типов....62
Указатели 64
Объекты....65

Процедуры и функции 65
О подпрограммах в Object Pascal....65
Процедуры 67
Функции 68
Рекурсия....70
Использование параметров 71
Области видимости 74
Видимость в модулях....76
Некоторые стандартные функции 78
Функции в действии....80

ЧАСТЬ II. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ....86

Введение в ООП 86
Суть современного программирования....86
Классы и объекты....88
Структура класса и видимость....89
Конструкторы и деструкторы 93
Методы объектов и оператор with....95
Классы в действии....97

Визуальные компоненты и ООП 100
Библиотека VCL....101
Класс TComponent....102
Свойства визуальных компонент....103
События....106
Методы визуальных компонент....108
Типы методов 109

Черчение, рисование и печать....111
Графика в Windows....111
Объект Canvas 112
Черчение фигур 114
Вывод на печать 116

Исключения и взаимодействие с API....119
Исключения и их классы 119
Обработка исключений....121
Глобальные объекты 123
Работа с INI-файлами....127

Работа с реестром Windows....131
Процедуры и функции стандартных диалогов....134
Обработка сообщений и Windows API....137

ЧАСТЬ III. ВИЗУАЛЬНОЕ ПРОГРАММИРОВАНИЕ....139

Работа с VCL в среде Delphi....139
Работа с визуальным редактором....139
Формы в Delphi....142
Класс TForm....144
Приложения SDI и MDI....148

Стандартные компоненты Delphi....150
Кнопки....150
Надписи....152
Поля ввода текста....153
Многострочный редактор и строки....156
Класс TStringList....159
Списки....161
Переключатели....164
Панели и группы....167
Меню....169
Коллекция действий....173

Компоненты 32-разрядного интерфейса....175
Коллекция картинок....175
Индикатор выполнения....176
Ползунок....178
Реверсивный счетчик UpDown....180
Редактор горячих клавиш....181
Компоненты для работы с датами....182
Строка состояния....185
Панель инструментов....187
Элементы с вкладками....190
Прочие компоненты Win32 и стиль Windows XP....192

Стандартные диалоги и редактор RTF....195
Диалоги работы с файлами....195
Диалоги печати....198
Диалог шрифта....200
Диалог цвета....203
Редактор RTF....205
Пример текстового редактора....208
Диалоги поиска и замены....214

Работа с файлами и мультимедиа....218
Списки файлов и каталогов....218
Файловые компоненты – списки....220

Отображение графических изображений 221
Классы TPicture и TGraphic 223
Пример программы просмотра графики 226
Компонент MediaPlayer 228
Компоненты Timer и PaintBox 231
Современные файловые компоненты 235

Дополнительные компоненты 238
Кнопки 238
Таблицы 239
Простейший табличный редактор 242
Другие дополнительные компоненты 248
Компоненты ActiveX 251
Установка компонентов 252

ЧАСТЬ IV. БАЗЫ ДАННЫХ 255

Теория баз данных 255
Введение в базы данных 255
Таблицы 256
Ключи и индексы 258
Организация данных и связь между таблицами 260
Целостность БД и транзакции 261
Язык SQL 263

Delphi и базы данных 264
Типы БД в Delphi 264
Инструменты для работы с БД 266
BDE и BDE Administrator 268
Создание таблиц в Database Desktop 270
Поддержка BDE в VCL 273
Альтернативы BDE 276

Компоненты доступа и представления данных 277
Доступ к данным 277
Таблица DB Grid 279
Навигация по таблице данных 281
Представление отдельных полей данных 283
Компонент DBCtrlGrid 284
Связывание данных в таблицах 285
Компоненты синхронного просмотра 287
Модуль хранения компонентов данных 289

Работа с данными 290
Состояния и режимы набора данных 290
Поля и класс TField 292
Типы полей и типы данных 294
Сортировка 296

Навигация 298
Фильтрация 300
Поиск 301
Редактирование 303
Добавление и удаление 305
Простое приложение БД 306

Запросы 312
Язык SQL 312
Команда SELECT 313
Некоторые другие команды SQL 319
Компонент запроса Query 321
Особенности работы с запросами 322

Отчеты 325
Отчеты в Delphi 325
Основы Rave Reports 328
Среда Rave Reports 329
Компоненты Rave Reports для создания отчетов 332
Компоненты Rave Reports в VCL 334
Пример создания отчета 336

Вместо заключения 339